COMPUTING SCIENCE

A Comparison of Formalisms for Modelling and Analysis of Dynamic Reconfiguration of Dependable Systems

Anirban Bhattacharyya, Andrey Mokhov and Ken Pierce

TECHNICAL REPORT SERIES

No. CS-TR-1462 April 2015

TECHNICAL REPORT SERIES

No. CS-TR-1462 April, 2015

A Comparison of Formalisms for Modelling and Analysis of Dynamic Reconfiguration of Dependable Systems

A.Bhattacharyya, A. Mokhov, K. Pierce

Abstract

This paper uses a case study to evaluate three formalisms of different kinds for their suitability for the modelling and analysis of dynamic reconfiguration of dependable systems. The reconfiguration of an office workflow for order processing is described. The requirements on the workflow's reconfiguration and general reconfiguration requirements are defined. The workflow is modelled using the Vienna Development Method (VDM), conditional partial order graphs (CPOGs), and the basic Calculus of Communicating Systems for dynamic process reconfiguration (basic CCSdp), and verification of the requirements is attempted using the models. The formalisms are evaluated according to their ability to model the reconfiguration, to verify the workflow's reconfiguration requirements, and to meet the general reconfiguration requirements.

Printed and published by Newcastle University,
Computing Science, Claremont Tower, Claremont Road,
Newcastle upon Tyne, NE1 7RU, England.

Bibliographical details

BHATTACHARYYA, A; MOKHOV, A; PIERCE, K;
A Comparison of Formalisms for Modelling and Analysis of Dynamic Reconfiguration of Dependable Systems
[By] A. Bhattacharyya, A. Mokhov, K. Pierce
Newcastle upon Tyne: Newcastle University: Computing Science, 2015.

(Newcastle University, Computing Science, Technical Report Series, No. CS-TR-1462)

Added entries

NEWCASTLE UNIVERSITY
Computing Science. Technical Report Series. CS-TR-1462

Abstract

This paper uses a case study to evaluate three formalisms of different kinds for their suitability for the modelling and analysis of dynamic reconfiguration of dependable systems. The reconfiguration of an office workflow for order processing is described. The requirements on the workflow's reconfiguration and general reconfiguration requirements are defined. The workflow is modelled using the Vienna Development Method (VDM), conditional partial order graphs (CPOGs), and the basic Calculus of Communicating Systems for dynamic process reconfiguration (basic CCSdp), and verification of the requirements is attempted using the models. The formalisms are evaluated according to their ability to model the reconfiguration, to verify the workflow's reconfiguration requirements, and to meet the general reconfiguration requirements.

About the authors

Anirban Bhattacharyya is a Research Associate in the School of Computing Science at Newcastle University. He received his B.Sc. (Hons) in Mathematics from the University of London King's College in 1982, and his M.Sc. in Information Systems Engineering from South Bank Polytechnic in 1984. Subsequently, he worked on a variety of R & D projects: at MARI Advanced Microelectronics Ltd (1985 - 1989), he researched into the instrumentation of an object-oriented distributed system, specification of a distributed meta IPSE using Z, enterprise modelling, and a documentation system for IBCN software. At York University (1989 - 1992), he worked on a modelling framework for hard real-time systems. Subsequently, he developed a variety of database applications for accounting, HR and payroll. Anirban studied for his PhD in the School of Computing Science at Newcastle University under the supervision of Dr. John Fitzgerald; and his thesis, entitled "Formal Modelling and Analysis of Dynamic Reconfiguration of Dependable Systems", was published in January 2013. His research interests include modelling and verification of dynamically reconfigurable dependable real-time systems using process algebras and model checking. Currently, Anirban is extending structured occurrence nets (SONs) with time in order to support timed causal reasoning on the EPSRC project UNCOVER.

Andrey Mokhov is a Lecturer in Computer Engineering at the School of Electrical and Electronic Engineering, Newcastle University. In 2000-2005 he studied Computing Science at Kyrgyz-Russian Slavic University, Kyrgyzstan. After graduation he joined the Microsystems Research Group at Newcastle University as a PhD student, and he successfully defended his PhD dissertation, entitled "Conditional Partial Order Graphs", in 2009. His current research is focused on a new approach to formal design of concurrent hardware and software systems, based on the compact 'overlaid' representation of families of graphs.

Ken Pierce is a Research Associate in the School of Computing Science at Newcastle University, associated with the AMBER group (advanced model-based engineering) and the CPLab (cyber-physical systems lab). His main interests lie in model-based design and engineering of cyber-physical systems (CPSs), in particular, in developing methods and tools for collaborative modelling and co-simulation, fault tolerance, and design space exploration (DSE). Ken received his BSc (Hons) in Computer Science (Software Engineering) from Newcastle University in 2005, and studied for his PhD under the supervision of Prof. Cliff Jones. His thesis, entitled "Enhancing the Usability of Rely-Guarantee Conditions for Atomicity Refinement", was published in December 2009. He participated in the successful FP7 projects DESTECS and COMPASS between 2010 and 2014. He now works on two H2020 ICT-1 projects: INTO-CPS, a research project that is building a tool chain for CPS design; and CPSE Labs, an innovation project that forms a network of CPS "design centres" across Europe and provides cascading funding for focused experiments in CPS design.

Suggested keywords

DYNAMIC SOFTWARE RECONFIGURATION
WORKFLOW CASE STUDY
RECONFIGURATION REQUIREMENTS
FORMAL METHODS
VDM
CONDITIONAL PARTIAL ORDER GRAPHS
BASIC CCSDP

A Comparison of Formalisms for Modelling and Analysis of Dynamic Reconfiguration of Dependable Systems

Anirban Bhattacharyya, Andrey Mokhov, Ken Pierce

***Abstract*—This paper uses a case study to evaluate three formalisms of different kinds for their suitability for the modelling and analysis of dynamic reconfiguration of dependable systems. The reconfiguration of an office workflow for order processing is described. The requirements on the workflow's reconfiguration and general reconfiguration requirements are defined. The workflow is modelled using the Vienna Development Method (VDM), conditional partial order graphs (CPOGs), and the basic Calculus of Communicating Systems for dynamic process reconfiguration (basic CCS^{dp}), and verification of the requirements is attempted using the models. The formalisms are evaluated according to their ability to model the reconfiguration, to verify the workflow's reconfiguration requirements, and to meet the general reconfiguration requirements.**

Index Terms – dynamic software reconfiguration, workflow case study, reconfiguration requirements, formal methods, VDM, conditional partial order graphs, basic CCS^{dp}

I. Introduction

The next generation of dependable systems is expected to have significant evolution requirements [7]. Moreover, it is impossible to foresee all the requirements that a system will have to meet in future when the system is being designed [29]. Therefore, it is highly likely that the system will have to be redesigned (i.e. reconfigured) during its lifetime, in order to meet new requirements. Furthermore, certain classes of dependable system, such as control systems, must be dynamically reconfigured [19], because it is unsafe or impractical or too expensive to do otherwise. The dynamic reconfiguration of a system is defined as the change at runtime of the structure of the system – consisting of its components and their communication links – or the hardware location of its software components [3] or their communication links. This paper focuses on dynamic software reconfiguration, because software is much more mutable than hardware.

Existing research in dynamic software reconfiguration can be grouped into three cases (see Figure 1).

Case 1 is the near-instantaneous reconfiguration of a system, in which the duration of the reconfiguration interval is negligible in comparison to the durations of application actions. Any executing task in Configuration 1 that is not in Configuration 2 is aborted, which can leave data in a corrupted or inconsistent state. Alternatively, the reconfiguration is done at the end of the hyperperiod of Configuration 1 (i.e. the lowest common multiple of the periods of the periodic tasks in Configuration 1), which can result in a significant delay in handling the reconfiguration-triggering event. This is the traditional method of software reconfiguration, and is applicable to small, simple systems running on a uniprocessor.

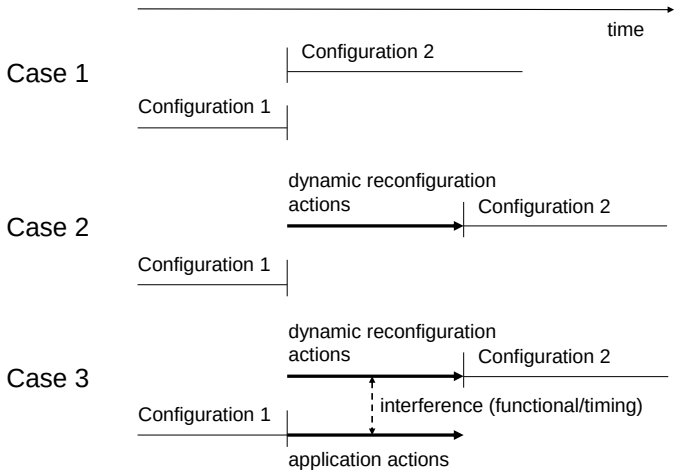


Figure 1: Dynamic reconfiguration cases.

Case 2 is the reconfiguration of a system in which the duration of the reconfiguration interval is significant in comparison to the durations of application actions, and any executing task in Configuration 1 that can interfere with a reconfiguration task is either aborted or suspended until the reconfiguration is complete. This is the most common method of software reconfiguration (see [41], [1], [5], and [20]), and is applic-

able to some large, complex, distributed systems. If the duration of the reconfiguration is bounded and the controlled environment can wait for the entire reconfiguration to complete, then the method can be used for hard real-time systems; otherwise, the environment can become irrecoverably unstable and suffer catastrophic failure if a time-critical service is delayed.

Case 3 is the reconfiguration of a system in which the duration of the reconfiguration interval is significant in comparison to the durations of application actions, and tasks in Configuration 1 execute concurrently with reconfiguration tasks. This method avoids aborting tasks and reduces the delay on the application due to reconfiguration, but it introduces the possibility of functional and timing interference between application and reconfiguration tasks. If the interference can be controlled, then this method is the most suitable for large, complex, distributed systems, including hard real-time systems, but it is also the least researched method of software reconfiguration. Existing research in Case 3 has focused on timing interference between application and reconfiguration tasks, and on achieving schedulability guarantees (for example, see [40], [42], [38], [36], [10], and [13]). There is little research on formal verification of functional correctness in the presence of functional interference between application and reconfiguration tasks (see [28] and [6]).

Therefore, there is a requirement for formal representations of dynamic software reconfiguration that can express functional interference between application and reconfiguration tasks, and can be analyzed to verify functional correctness. This paper makes a contribution towards meeting this requirement. Research shows that no single existing formalism is ideal for representing dynamic reconfiguration [44]. Therefore, we use three formalisms of different kinds: the Vienna Development Method (VDM, based on the state-based approach), conditional partial order graphs (CPOGs, in which graph families are used for verification of workflow and reconfiguration requirements), and the basic Calculus of Communicating Systems for dynamic process reconfiguration (basic CCS^{dp}, based on the behavioural approach), to produce representations of a case study, and evaluate how well the different representations meet the requirement.

The rest of the paper is organized as follows: Section II describes the case study, in which a simple office workflow for order processing is reconfigured, and defines the requirements on Configuration 1, Configuration 2, and on the reconfiguration of the workflow. Requirements on an ideal formalism for dynamic software reconfiguration are also identified, based on [3]. The reconfiguration of the workflow is modelled and analyzed using VDM (in Section III), CPOGs (in Section IV), and basic CCS^{dp} (in Section V). We have deliberately not used workflow-specific formalisms (such as [45], [2], and [14]) for two reasons. First, because of our lack of fluency in them; and second, because we believe the models should be produced using general purpose formalisms (if possible). The findings of the modelling and analysis exercise are discussed in Section VI.

II. Requirements

The case study described in this section involves the dynamic reconfiguration of a simple office workflow for order processing, which is a simplified version of real workflows commonly found in large and medium-sized organisations [9]. A preliminary version of this case study is given in [27]. The workflow consists of a network of several communicating activities, and the configuration of the workflow is the structure of the network. Initially, the workflow executes in Configuration 1 and must satisfy the requirements on Configuration 1. Subsequently, the workflow must be reconfigured through a process to Configuration 2 such that the requirements on the reconfiguration process and on Configuration 2 are satisfied.

We define the activities of the workflow, the requirements on the two configurations and on the reconfiguration process, the design of the workflow to be reconfigured, and then identify requirements on an ideal formalism for the modelling and analysis of dynamic software reconfiguration, based on [3].

A. Activities of the Office Workflow for Order Processing

The workflow initially contains the following activities:

1) `Order Receipt`: an order for a product is received from an existing customer. The order includes the customer's identifier and the product's identifier. An evaluation of the order is initiated to determine whether or not the order is viable.
2) `Evaluation`: in evaluating the order, the product identity is used to perform an inventory check on the availability of the product. The customer identity is used to perform a credit check on the customer. If both checks are positive, the order is accepted; otherwise, the order is rejected.

3) `Rejection`: if the order is rejected, a notification of rejection is sent to the customer and the workflow terminates.
4) If the order is accepted, the following activities are initiated:
   a) `Shipping`: the product is shipped to the customer.
   b) `Billing`: the customer is billed for the cost of the product ordered plus shipping costs.
   c) `Archiving`: the order is archived for future reference.
   d) `Confirmation`: a notification of successful completion of the order is sent to the customer.

The initial configuration of the workflow is Configuration 1, which must meet the following requirements (see Figure 2).

B. Requirements on Configuration 1 of the Workflow

For each order:

1) `Order Receipt` must be performed first. That is, it must begin before any other activity.
2) `Evaluation` must be performed second.
3) If the output of `Evaluation` is negative, `Rejection` must be the third and final activity of the workflow.
4) If the output of `Evaluation` is positive, the following conditions must be satisfied:
   a) `Shipping` must be the third activity to be performed.
   b) `Billing` must be the fourth activity to be performed.
   c) `Archiving` must be the fifth activity to be performed.
   d) After the completion of `Archiving`, `Confirmation` must be the sixth and final activity to be performed.
   e) The customer must not receive more than one shipment of an order (safety requirement).
5) Each activity must be performed at most once.
6) The order must be either rejected or satisfied (liveness requirement).
7) The workflow must terminate.

After some time, the management of the organization using the workflow decides to change it in order to increase sales and provide a faster service. The new configuration of the workflow is Configuration 2, which must meet the following requirements (see Figure 3).

C. Requirements on Configuration 2 of the Workflow

For each order:

1) `Order Receipt` must be performed first.
2) `Evaluation`: in evaluating the order, the product identity is used to perform an inventory check on the availability of the product. If the inventory check fails, an external inventory check is made on the suppliers of the product. The customer identity is used to perform a credit check on the customer. If either the inventory check or the supplier check is positive, and the credit check is positive, the order is accepted; otherwise, the order is rejected.
3) `Evaluation` must be performed second.
4) If the output of `Evaluation` is negative, `Rejection` must be the third and final activity of the workflow.
5) If the output of `Evaluation` is positive, the following conditions must be satisfied:
   a) `Billing` and `Shipping` must be the third set of activities to be performed.
   b) `Billing` and `Shipping` must be performed concurrently.
   c) After the completion of both `Billing` and `Shipping`, `Archiving` must be the fourth and final activity to be performed.
   d) The customer must not receive more than one shipment of an order (safety requirement).
6) Each activity must be performed at most once.
7) The order must be either rejected or satisfied (liveness requirement).
8) The workflow must terminate.

In order to achieve a smooth transition from Configuration 1 to Configuration 2 of the workflow, the process of reconfiguration must meet the following requirements.

D. Requirements on Reconfiguration of the Workflow

1) Reconfiguration of a workflow should not necessarily result in the rejection of an order.
   In some systems, executing activities of Configuration 1 are aborted during its reconfiguration to Configuration 2 (see Case 2 in Figure 1). The purpose of this requirement is to avoid the occurrence of Case 2 and ensure the occurrence of Case 3.
2) Any order being processed that was accepted **before** the start of the reconfiguration must satisfy all the requirements on Configuration 2

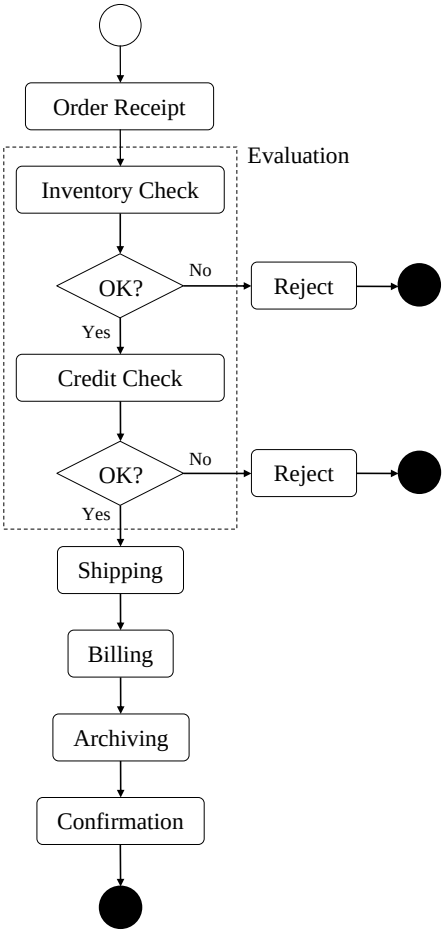


Figure 2: Flow chart of the requirements on Configuration 1.

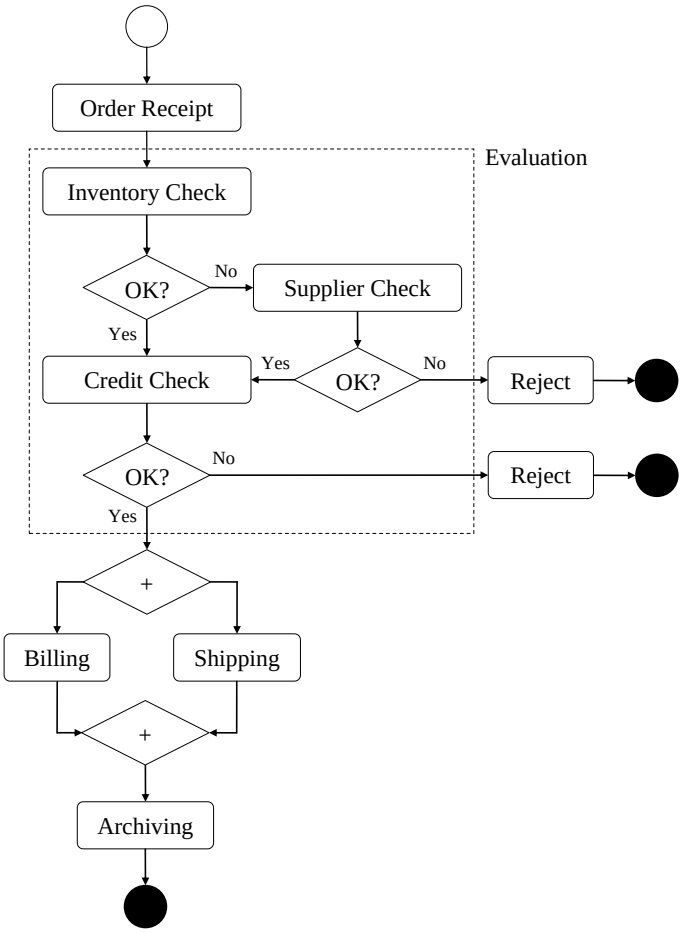


Figure 3: Flow chart of the requirements on Configuration 2.

(if possible); otherwise, all the requirements on Configuration 1 must be satisfied.
3) Any order accepted **after** the start of the reconfiguration must satisfy all the requirements on Configuration 2.
4) The reconfiguration process must terminate.

E. Design of the Workflow and its Reconfiguration

The reconfiguration of the workflow depends on the workflow's design and implementation. There are at least four possible designs for the workflow:

1) There is at most one workflow, and the workflow handles a single order at a time.
   The workflow is sequential: after an order is received, the thread performs a sequence of actions, with two choices at `Evaluation`. After the order has been processed, the thread is ready to receive a new order. This design corresponds to a cyclic executive.
2) There is at most one workflow, and the workflow can handle multiple orders at a time.
   The workflow is mainly concurrent: after an order is received, the thread forks internally into concurrent threads, such that different threads perform the different activities of the workflow – although the requirements on the configurations severely restrict the degree of internal concurrency of the workflow – and each thread performs the same activity for different orders.
3) There can be multiple workflows, and each workflow handles a single order.
   After an order is received, the thread forks into two threads: one thread processes the order sequentially – as in Design 1 – but terminates after the order has been processed; the other thread waits to receive a new order.
4) There can be multiple workflows, and each workflow can handle multiple orders at a time.
   This design is a complex version of Design 2 with multiple workflows.

We choose to model Design 3 because it has a mixture of sequential and concurrent processing, and is simple and realistic.

There are three possible designs for the reconfiguration of the workflow:

1) The reconfiguration consists of a single action.
2) The reconfiguration consists of multiple actions performed sequentially.
3) The reconfiguration consists of multiple actions performed concurrently.

We choose to model Design 3 because it provides the maximum scope for interference between application and reconfiguration actions.

F. Requirements on an Ideal Formalism for Dynamic Software Reconfiguration

No single existing formalism is ideal for the modelling and analysis of dynamic software reconfiguration [44]. However, it is possible to identify core requirements that an ideal formalism must meet, and use these requirements to evaluate the formalisms examined in this paper. The requirements are taken from [3] and are justified briefly.

Dynamic Reconfiguration Requirements

1) It should be possible to model, and to identify instances of, software components and tasks, and their communication links.
   Many dependable systems use multiple instances of a software component to provide fault tolerance. The dynamic reconfiguration of the component involves the selective reconfiguration of its instances, which is facilitated by the use of instance identifiers.
2) It should be possible to model the creation, deletion, and replacement of software components and tasks, and the creation and deletion of their communication links.
   These are the fundamental operations used to change the software structure of a system.
3) It should be possible to model the relocation of software components and tasks on physical nodes.
   Software relocation helps to implement load balancing, which is used to improve performance and reliability in cloud computing. Thus, software relocation helps to increase the dependability of cloud computing.
4) It should be possible to model state transfer between software components and between tasks.
   In dependable systems with state, state transfer helps to implement Case 2 of dynamic reconfiguration (see Figure 1) and to implement software relocation.
5) It should be possible to model both planned and unplanned reconfiguration.
   Planned reconfiguration is reconfiguration that is incorporated in the design of a system. Unplanned reconfiguration is reconfiguration that is **not** incorporated in the design of a system, which is relevant for legacy systems and for the evolution of systems.
6) It should be possible to model the functional interference between application tasks and reconfiguration tasks.
   This is the main modelling requirement in Case 3 of dynamic reconfiguration (see Figure 1), and is an outstanding research issue.
7) It should be possible to express and to verify the functional correctness requirements of application tasks and reconfiguration tasks.
   This is the main verification requirement of dynamic reconfiguration, and is an outstanding research issue in Case 3.

General Requirements

1) It should be possible to model the concurrent execution of tasks.
   Concurrency can cause functional interference between tasks, and thereby affect the functional correctness of a task, and it is a feature of many dependable systems. Therefore, it should be modelled.
2) It should be possible to model state transitions of software components and tasks.
   State affects the functionality of a task, and thereby affects the functional correctness of the task, and it is a feature of most dependable systems. Therefore, it should be modelled.
3) The formalism should be as terse as possible, in order to facilitate its use.
4) The formalism should be supported by tools; otherwise, it will not be used by software engineers.

III. VDM

The Vienna Development Method (VDM) is a state-based formal method that was originally designed in the 1970s to give semantics to programming languages [18]. Since then it has been used widely both in academia and industry to define specifications of software systems. It is well-suited to formalise requirements and natural language specifications and

to find defects. For example, the "FeliCa" contactless card technology, which is widely used in Japan, was developed using VDM [12]. A specification was constructed in VDM that revealed a large number of defects (278) in the existing natural-language requirements and specifications. The VDM specification was used to generate test cases and as a reference when writing the C++ code that was eventually deployed to millions of devices.

The VDM Specification Language (VDM-SL) was standardised as ISO/IEC 13817-1 in 1996 [16]. Developments beginning in the 1990s extended the language to cover object-orientation (VDM++ [11]) and later to include abstractions for modelling real-time embedded software (VDM-RT [43]). All three dialects are supported by two robust tools, the commercial VDMTools [21] and the open-source Overture[1] tool [22]. These tools offer type checking for VDM models, a number of analysis tools such as combinatorial testing [23], and interpretation of an executable subset of VDM that allows models to be simulated. By connecting a graphical interface to an executable model, it is also possible to animate specifications [12], thereby allowing non-specialists to gain an understanding of the system described by the specification through interaction and interrogation of the model.

The models of our case study were developed in VDM-SL (using the Overture tool) and the remainder of this section uses that dialect. As part of the standardisation process, a full denotational semantics has been defined for VDM-SL [24], as well as a proof theory and comprehensive set of proof rules [4]. Proofs in VDM typically verify internal consistency or are proofs of refinement [17].

We proceed as follows: the VDM formalism is described in more detail in Section III-A. The modelling of the case study is described in Section III-B, the analysis of the model is described in Section III-C, possible extensions to the model are described in Section III-D, and an evaluation of the model and formalism for describing reconfiguration is given in Section III-E.

A. Formalism

Specifications in VDM-SL are divided into modules, where each module contains a set of definitions. Modules can export definitions to, and import definitions from, other modules in the model. The definitions in a module are divided into distinct sections and preceded by a keyword. Definitions can include types, values, functions, state, and operations. Figure 4 shows an empty VDM-SL module divided into sections. We give an overview of the definitions below. Further detail is introduced as required during explanation of the model.

```
module MyModule

exports ...
imports ...

definitions

types
...

functions
...

state ... of
...
end

operations
...

values
...

end MyModule
```

Figure 4: Empty VDM-SL module specification.

A key part of the VDM-SL language is the powerful type system. VDM-SL contains a number of built-in scalar types including **boolean**s, numeric types, and characters. Non-scalar types include **set**s, **seq**uences, and **map**s. Custom types can be defined in the **types** section, based on built-in types and including type unions and record types with named fields. Custom types can be restricted by invariants and violations of these **inv**ariants can be flagged during interpretation of the models.

Functions are pure and have no side effect. They can be defined implicitly (by pre- and post-conditions) or explicitly. Explicit functions can also be protected with a pre-condition. Only explicit functions can be executed. *State* allows one or more global variables to be defined for the module. An invariant can be defined over the state to restrict its values. Again, invariant violations can be flagged during interpretation by the tool. *Operations* are functions that are additionally able to read and write state variables. Therefore, operations can have side effects. Like functions, operations can be defined implicitly or explicitly. *Values* define constants that can be used in functions and operations.

B. Modelling

In creating a specification that meets the workflow requirements given in Section II, two approaches are

[1] http://www.overturetool.org/

possible in the VDM 'idiom'. The first is to build a data model that captures an order and its status, with operations and pre-conditions ensuring that only valid transitions between statuses are possible (e.g. order receipt to inventory check). Such a model could also include details of customers and suppliers. The second approach, and the one selected for this study, is to model the entire workflow and build an interpreter to execute and reconfigure it.

The second approach more closely matches the style of the requirements. However, the approaches are not orthogonal; adding the data model would complement the workflow model and the resulting specification is likely to be closer to any code to be written in implementing the order system. We return to the first approach in Section III-D, giving an example of how the model could be extended to incorporate a data model for orders.

The following subsections explain the modelling process in detail. First, a set of types is defined that can capture the two configurations. Next, an interpreter is created that can 'execute' a workflow and can perform various tests. In order to test all possible paths through a configuration, a method for setting the outcome of external choices (inventory check, credit check, and supplier check) is included. Finally, a reconfiguration operation is added that initially allows reconfiguration to any arbitrary configuration. The model is then extended with an invariant and pre-condition to permit only safe reconfiguration.

The model is split into three modules: `Configurations`, containing workflow definitions; `Interpreter`, containing operations to interpret and reconfigure workflows; and `Test`, which defines test cases for the model.

Workflows and Traces

The `Configurations` module defines types that can represent the workflows from Figures 2 and 3 and are used by the interpreter (shown later). It also defines two constants that instantiate these workflows, a type to represent a trace of a workflow execution, and some useful auxiliary functions. The types, functions, and values of this module are made available to both the other modules using the **`exports all`** declaration:

```
module Configurations

exports all

definitions

...

end Configurations
```

The types for capturing workflows and their traces are built around a core type called `Action`. This type enumerates all possible activities in a workflow (and therefore also in a trace). It requires that any value assigned to an action must be exactly one of the nine listed values. The *type union* is defined using the pipe (|) operator, and the individual values are *quote* types (basic values that can only be compared for equality):

```
-- actions in the workflow
Action = <OrderReceipt> | <InventoryCheck> | <Reject>
       | <CreditCheck> | <SupplierCheck> | <Shipping>
       | <Billing> | <Archiving> | <Confirmation>;
```

Based on this type, we define a trace as a sequence of events recording either the occurrence of an action, or a special <TERMINATE> event indicating successful completion of a workflow. The invariant on `Trace` states that if a termination event occurs, it must occur at the end (i.e. if it appears in the trace and is not the only element, then it does not appear in the first $n-1$ elements):

```
-- record of an action or termination
Event = Action | <TERMINATE>;

-- trace of events
Trace = seq of Event
inv t == (<TERMINATE> in set elems t and len t > 1) =>
  <TERMINATE> not in set elems t(1, ..., len t - 1);
```

To define a workflow type, it is necessary to allow an order for actions to be specified; this could be done with a sequence. However, in this study a recursive definition is used based around a type called `Element`:

```
-- workflow elements
Element = [Simple | Branch | Par];
```

This definition states that an `Element` can be one of three other types (expanded below): `Simple` represents a single transition such as order receipt to inventory check; `Branch` represents an *OK?* choice, such as the credit check; and `Par` represents parallel composition. The square brackets make the type *optional*, meaning that it can take a fourth special value (**`nil`**) that represents termination (the black circles in Figures 2 and 3). These three types are defined as follows:

```
-- a simple element
Simple :: a : Action
          e : Element;

-- a conditional element
Branch :: a : Action
          t : Element
          f : Element;

-- parallel elements
Par :: b1 : Action
       b2 : Action
        e : Element;
```

The above three definitions are *record* types, that is, compound types with named elements. Each type contains one or more actions to be executed, and one or more elements that follow this action. Therefore, the definitions are recursive, and the recursion is terminated by a **nil** value at each leaf. A simple workflow (called T) that rejects all orders could be defined as:

```
values

T = mk_Simple(<OrderReceipt>, mk_Simple(<Reject>, nil))
```

Notice that the mk_ keyword is a *constructor* used to instantiate values of record types. They are essentially (automatically defined) functions that construct a record with the parameters being assigned, in order, to named elements.

The Configurations module also defines two auxiliary functions that are useful for invariants and preconditions. The first (prefixof) determines if one trace if a prefix of another and the second (tracesof) recursively computes all elements of a trace:

```
-- true if a is a prefix of b, false otherwise
prefixof: Trace * Trace -> bool
prefixof(a, b) == ...

-- compute all traces of an element
tracesof: Element -> set of Trace
tracesof(el) == ...
```

The Element type could be used as-is to represent workflow configurations, but it is not restricted in any way. For example, it can contain repeated actions (i.e. billing or shipping twice). Therefore we introduce a Workflow type with an invariant that prevents duplicates (by checking that for all traces of the workflow, the cardinality of the set of events in the trace is the same as the length of the trace):

```
-- a workflow
Workflow = Element
inv w == forall tr in set tracesof(w) &
  card elems tr = len tr;
```

Finally, the module defines values (constants) that describe the two configurations in the requirements. These are shown as Configuration1 and Configuration1 in Figure 5.

Interpreter

The Interpreter module allows a workflow to be interpreted. The module exports its definitions so that they can be accessed by the Test module, and it imports required type definitions from the Configurations module.

```
-- first configuration
Configuration1: Workflow =
  mk_Simple(<OrderReceipt>,
    mk_Branch(<InventoryCheck>,
      mk_Branch(<CreditCheck>,
        mk_Simple(<Shipping>,
          mk_Simple(<Billing>,
            mk_Simple(<Archiving>,
              mk_Simple(<Confirmation>, nil)
            )
          )
        ),
        mk_Simple(<Reject>, nil)
      ),
      mk_Simple(<Reject>, nil)
    )
  );

-- second configuration
Configuration2: Workflow =
    mk_Simple(<OrderReceipt>,
        mk_Branch(<InventoryCheck>,
            mk_Branch(<CreditCheck>,
                mk_Par(<Billing>, <Shipping>,
                    mk_Simple(<Archiving>, nil)),
                mk_Simple(<Reject>, nil)
            ),
            mk_Branch(<SupplierCheck>,
                mk_Branch(<CreditCheck>,
                    mk_Par(<Billing>, <Shipping>,
                        mk_Simple(<Archiving>, nil)),
                    mk_Simple(<Reject>, nil)
                ),
                mk_Simple(<Reject>, nil)
            )
        )
    );
```

Figure 5: Workflow configurations represented in VDM-SL.

```
module Interpreter

exports all

imports from Configurations types Workflow, Trace, ...
```

The state of the module records the trace of the interpretation so far (trace) and the remaining workflow to be interpreted (workflow). A state is similar to a record type and is defined in a similar manner:

```
-- interpreter state
state S of
     trace : Trace
  workflow : Workflow
init s == s = mk_S([], nil)
end;
```

The above state definition contains an init clause that gives initial values to both components of the state (they are both 'empty'). An invariant can also be defined with an inv clause, but an invariant is not required at this point. The module provides operations to set (and reset) the state of the interpreter, to step through execution of a workflow or interpret it in a single step, and to access the current value of the trace. Operations for reconfiguration are also included and are described below (see *Reconfiguration*).

An operation called `Init` is used to prime (set and reset) the interpreter with a workflow passed as a parameter and an empty trace:

```
Init: Workflow ==> ()
Init(w) == (
    trace := [];
    workflow := w
);
```

The basic operation of the interpreter is to move an action from the head of `workflow` and append it to the end of the trace. Once the workflow is empty, the <TERMINATE> element is added to the trace and the interpretation ends. Since there is no data model underlying the workflow, no additional work is done when moving an action from the workflow to the trace. However, an extension is considered towards this in Section III-D.

The absence of a data model also means that the external choices in the workflow (the inventory check, credit check, and supplier check) must be made in some other manner. The main analysis method for this model is testing (described in Section III-C). Therefore, it is desirable to be able to control these external choices to ensure test coverage. In order to do this, a Choices type is introduced, which is a mapping from (choice) actions to Boolean. The invariant ensures that the domain of the map is exactly the set of actions that represent external choices:

```
-- collapse probabilities
Choices = map Action to bool
inv c == dom c =
  {<InventoryCheck>, <CreditCheck>, <SupplierCheck>}
```

For example, a run of the workflow where there is sufficient inventory and sufficient credit can be achieved using the following choices (in this case a supplier will not be needed):

```
-- all branches true
NoProblems = {
  <InventoryCheck> |-> true,
   <SupplierCheck> |-> true,
     <CreditCheck> |-> true
};
```

The `Test` module defines values for a total of five combinations of choices, which are sufficient to test all branches of the two workflows.

The `Interpreter` module defines two operations that perform the interpretation, `Step` and `Interpret`, with the following signatures:

```
-- perform a single step of the interpreter
Step: Choices ==> Event

-- interpret workflow in one go
Interpret: Choices ==> ()
```

The `Step` operation performs a single step of interpretation, updating the trace and moving to the next step of the workflow. This operation selects the outcome of `Branch` actions based on the `Choices` passed as a parameter, and the order of execution of actions in `Par` elements are selected randomly leading to interleaving of the actions. The `Step` operations returns the last event that occurred, which is used for reconfiguration (described below).

The `Interpret` operation uses `Step` operation to run through a workflow and produce a full trace. The **`let`** expression is used to ignore the value returned by `Step`, since it is not needed for a simple interpretation run:

```
-- interpret workflow in one go
Interpret: Choices ==> ()
Interpret(c) == (
    while workflow <> nil do
      let - = Step(c) in skip;
    trace := trace ^ [<TERMINATE>]
);
```

Reconfiguration

Reconfiguration is achieved by replacing the current workflow in the state by another workflow during interpretation. We consider the case of a single thread of interpretation moving from some point in Configuration 1 to an appropriate point in Config2, with extensions discussed later. The following operation is defined in the `Interpreter` module that replaces the workflow in the state by the workflow passed as a parameter to the operation. The point at which this operation is called, and the workflow passed to the operation, are left to the caller:

```
-- reconfigure, replacing current workflow
Reconfigure: Workflow ==> ()
Reconfigure(w) ==
  workflow := w;
```

In this unprotected form, the calling thread is able to make arbitrary changes to the workflow, resulting in traces that do not meet the requirements described earlier. For example, double billing a customer by reconfiguring to a workflow with a <Billing> element after billing had already occurred.

This is avoided by adding an invariant to the state that disallows configurations that could generate illegal traces. Additionally, a pre-condition is added to the `Reconfigure` operation to protect the invariant, ensuring that the operation only processes valid reconfigurations. Ideally, invariants should be protected by pre-conditions on operations in this fashion, such that invariants form a “last line of defence”.

It is a requirement that traces produced by the interpreter must be traces of Configuration 1 or Configuration 2. Therefore, the invariant states that, given the current trace (which may be empty), the remaining workflow can only produce traces valid under Configuration 1 or Configuration 2. Similarly, the pre-condition checks that the new workflow can only produce valid traces Configuration 2 (since we currently only consider reconfigurations from Configuration 1 to Configuration 2). With the pre-condition added, the operation is defined as follows:

```
-- reconfigure, replacing current workflow
Reconfigure: Workflow ==> ()
Reconfigure(w) ==
  workflow := w
pre forall t in set
      {trace ^ tr | tr in set tracesof(w)} &
      (exists x in set tracesof(Config2) &
          prefixof(t, x));
```

This pre-condition makes use of the `tracesof` and `prefixof` auxiliary functions. It checks that for all traces in the set of traces produced by appending the possible traces of the new configuration to the current trace, there exists at least one trace of Config2 that the trace is a prefix of. Therefore, all traces that could occur after reconfiguration are valid under Configuration 2. The invariant is defined similarly, except that both Configuration 1 and Configuration 2 are checked.

C. Analysis

The `Test` module defines operations that test both configurations with the five combinations of external choices. These operations call the operations of the `Interpreter` module (using the operator ‘) and output the trace, which can then be printed to the console in Overture. For example, the operation that tests `Config1` with the `NoProblems` choices is defined below. This operation initialises the interpreter with `Config1`, runs the interpreter and returns the completed trace:

```
-- Test Config1 / NoProblems
Config1NoProblems: () ==> Trace
Config1NoProblems() == (
  Interpreter‘Init(Config1);
  Interpreter‘Interpret(NoProblems);
  return Interpreter‘GetTrace()
);
```

When printed to the console, the output of the `Config1NoProblems` operation shows the following trace:

```
Test‘Config1NoProblems() =
  [<OrderReceipt>, <InventoryCheck>, <CreditCheck>,
   <Shipping>, <Billing>, <Archiving>, <Confirmation>,
   <TERMINATE>]
```

In order to test the reconfiguration operation, and demonstrate the outcome of a valid and invalid reconfiguration request, the test module defines an operations called `TestReconfig` with the following signature:

```
TestReconfig: Choices * Action * Workflow ==> ()
TestReconfig(c, rp, w) == ...
```

In addition to the `Choices` required for interpretation (`c`), the operation takes an action (`rp`) and a workflow (`w`) as parameters. The operation initialises the interpreter, then steps through the interpretation until the action `rp` is seen, then attempts to reconfigure the interpreter to the workflow `w`. If the reconfiguration is valid under the requirements, the final trace will be printed. Otherwise, a message is printed stating that the reconfiguration is invalid (and that the pre-condition would fail if interpretation continued).

Using `TestReconfig`, two operations are defined that demonstrate a valid and invalid reconfiguration respectively. The first, `TestReconfigSuccess`, reconfigures from Config1 to Config2 after the inventory check (where there is no inventory in stock, so a supplier check is performed). This is a valid reconfiguration, and the console output is as follows:

```
[<OrderReceipt>, <InventoryCheck>]
Reconfiguring Config1 to Config2...
[<OrderReceipt>, <InventoryCheck>, <SupplierCheck>,
 <CreditCheck>, <Billing>, <Shipping>, <Archiving>,
 <TERMINATE>]
```

The second operation, `TestReconfigFail`, attempts to reconfigure from Config1 to the parallel composition of shipping and billing in Config2 after shipping has already occurred. This is an invalid reconfiguration, since shipping will occur twice. The output on the console is as below:

```
[<OrderReceipt>, <InventoryCheck>, <CreditCheck>,
 <Shipping>]
Reconfiguring Config1 to Config2...
These potential traces are not valid under Config2:
* [<OrderReceipt>, <InventoryCheck>, <CreditCheck>,
   <Shipping>, <Billing>, <Shipping>, <Archiving>]
* [<OrderReceipt>, <InventoryCheck>, <CreditCheck>,
   <Shipping>, <Shipping>, <Billing>, <Archiving>]
```

Notice that the Overture tool terminates execution with an error due to pre-condition failure:

```
Reconfiguration could generate invalid traces; pre-
condition will fail.

Error 4071: Precondition failure: pre_Reconfigure in
‘Interpreter’
```

D. Extensions

The model describes an interpreter with a single thread. Therefore, interference is not considered bey-

ond the non-deterministic execution of parallel compositions. To allow multiple threads of control, the state of the interpreter must be extended to allow a set of workflows (and their associated traces) to be defined. The reconfiguration operation must then be extended to reconfigure each thread in turn. This would allow the model to exhibit concurrent application and reconfiguration actions (Case 3 in Figure 1).

However, notice that actions are currently atomic, so there is no way to *abort* actions. To extend the model to allow this, a notion of beginning and completing actions would be required. This could be achieved either by having 'begin' and 'end' forms of each action, or by defining a 'current action' in the state, which is placed there and removed at a later state. If reconfiguration occurs between the beginning and end of an action, or if there is a current action present in the state, then an abort occurs.

To extend the model to allow reconfiguration back to Configuration 1 from Configuration 2, the pre-condition on `Reconfigure` needs to be relaxed to permit future traces from both configurations (not just Configuration 2 in the current model). To extend the model to add further configurations, a few steps are necessary. First, the configuration must be defined as a value in the `Configurations` module (for example, `Configuration3`). Tests for this new configuration should be added to the `Test` module and executed. Finally, the pre-condition and invariant must be extended to consider traces of the new configuration to be valid. This is simple if it is acceptable to switch between any configuration at any time. However, the definitions would be more complicated if there were restrictions on reconfiguration. For example, if there are 'points of no return' in between different configurations.

In Section III-B, extending the current model with a model of data was suggested. This extension represents an augmentation of the current workflow models with the data and operations necessary to allow customers to place orders. This could include data types for representing orders, such as the following:

```
CustId = token;
OrderId = token;

Order ::      custid : CustId
         inventoryOK : [bool]
            creditOK : [bool]
              accept : [bool];

state Office of
  orders : map OrderId to Order
end
```

The above defines identifiers for customers and orders using **`token`** types (a countably infinite set of distinct values that can be compared for equality and inequality). The `Order` type is a record that identifies a customer and the status of the order: whether or not the checks have been passed, and whether the order is accepted. The state of the model stores all orders in a map.

To continue this model, operations should be defined to receive and evaluate orders, and to accept or reject them, then to notify the customer, to bill, ship and finally archive. Each should manipulate the data model and be protected by pre-conditions to ensure consistency and make explicit any assumption about the system.

```
EvaluateOrder(oid:OrderId)
ext wr orders
pre oid in set dom orders and
    orders(oid).inventoryOK = nil and
    orders(oid).creditOK = nil and
    orders(oid).accept = nil
post exists iOK, cOK : bool & orders = orders~ ++
       {oid |-> mu(orders(oid),
         inventoryOK |-> iOK, creditOK |-> cOK)};
```

This extended data model could then be connected to the existing interpreter, such that when elements of the workflow are executed, calls are made to the operations that manipulate the data model. The extended model would demonstrate whether it was possible to build an actual order system that met the requirements, particularly when new configurations are introduced. The data and operations could then be used as a specification during implementation in some programming language.

E. Evaluation

We now identify the strengths and weaknesses of VDM, based on observations in modelling the case study.

Strengths

The functionality of VDM allows workflows to be captured as data types, including their actions, configurations, and traces. Invariants allow these types to be restricted as necessary to capture properties of the data. For example, to prevent definition of workflows with duplicated actions.

VDM allows an interpreter to be defined in order to test workflows and check their requirements. Concurrency is captured through interleaving in the interpreter, while both planned and unplanned reconfiguration are captured by allowing the state of the interpreter to be changed during execution. A pre-condition on the reconfiguration operation restricts

reconfigurations to only those that result in valid traces as described in the requirements.

VDM is supported by two industrial-strength tools (Overture and VDMTools) that are both under active development. These tools provide syntax highlighting and type checking facilities, and include interpreters that allow simulation of the workflow case study. The tools also provide more advanced features such as unit and combinatorial testing and proof obligation generation.

Although not explored in detail within this study, VDM would allow the data and functionality required in a full order system to be modelled (see Section III-D) and refined towards the final code that would be required of a real system and chosen programming language.

Weaknesses

Although modelling of the workflow case study was achieved, as a general purpose modelling language, VDM does not contain built-in abstractions or primitives for modelling processes or concurrency. This implies that features such as fine-grained concurrency or true parallelism are more difficult to achieve, since they must be modelled 'from scratch'.

The verification of the case study used simulation and testing, which is much weaker than model checking. The two VDM tools do not currently support model checking due to the generality of the formalism. While a proof theory exists for core VDM functionality, there is currently a lack of tool support for discharging proof obligations (although they can be generated automatically). In addition, the object-oriented and real-time extensions to VDM are currently not covered by the proof theory. However, this is an active area of research.

IV. Conditional Partial Order Graphs

Conditional partial order graphs (CPOGs) [34] were originally introduced for reasoning about properties of *families of graphs* in the context of asynchronous microcontroller design [33] and processors with reconfigurable microarchitecture [35]. In this paper, CPOGs are used to represent efficiently graph families in which each graph expresses a workflow *scenario* (i.e. a particular collection of outcomes of branching actions of a workflow) or a particular collection of requirements on workflow actions and their order. For example, the requirements on Configuration 1 of the case study workflow (shown in Figure 2) can be expressed using a family of three simple (i.e. branch-free) graphs (see Figure 6). We use the term *family* instead of the more general term *set* to emphasise the fact that the graphs are annotated with branch decisions, in this case:

- Inventory check OK: No
- Inventory check OK: Yes, credit check OK: No
- Inventory check OK: Yes, credit check OK: Yes

This can be expressed equivalently using the following predicates:

- ¬(InventoryCheck OK)
- (InventoryCheck OK) ∧ ¬(CreditCheck OK)
- (InventoryCheck OK) ∧ (CreditCheck OK)

where InventoryCheck and CreditCheck are the names of the two branch actions. Henceforth, we adopt the predicate notation.

The remainder of this section is organized as follows. In Section IV-A an axiomatic definition of CPOGs is given and basic workflow modelling primitives are introduced, which are then used in Section IV-B to model the case study workflow and to demonstrate how CPOGs can be used for workflow verification by reduction to the Boolean Satisfiability (SAT) problem. Dynamic reconfiguration is discussed in Section IV-C, and we show that CPOGs use true concurrency semantics when modelling functional interference between workflow and reconfiguration actions. We comment on our experience of automating the verification of the workflow requirements in Section IV-D, and give an overall evaluation of the CPOG-based approach in Section IV-E.

A. Axioms

The algebraic definition of CPOGs is outlined below, and was first introduced in [32] to provide superior compositionality and abstraction. Each element of the algebra represents a family of graphs, and is defined as follows:

Let $\mathcal{A}$ be the alphabet of names of actions (such as OrderReceipt and InventoryCheck in Figure 2) that can occur in a workflow. $\mathcal{A}$ is used to construct models of workflows and to define their requirements using the following axioms:

- The empty workflow is denoted by ε, that is, the empty family of graphs.
- A workflow consisting of a single action $a \in \mathcal{A}$ is denoted simply by a. It corresponds to a family consisting of a single graph that contains a single vertex a.

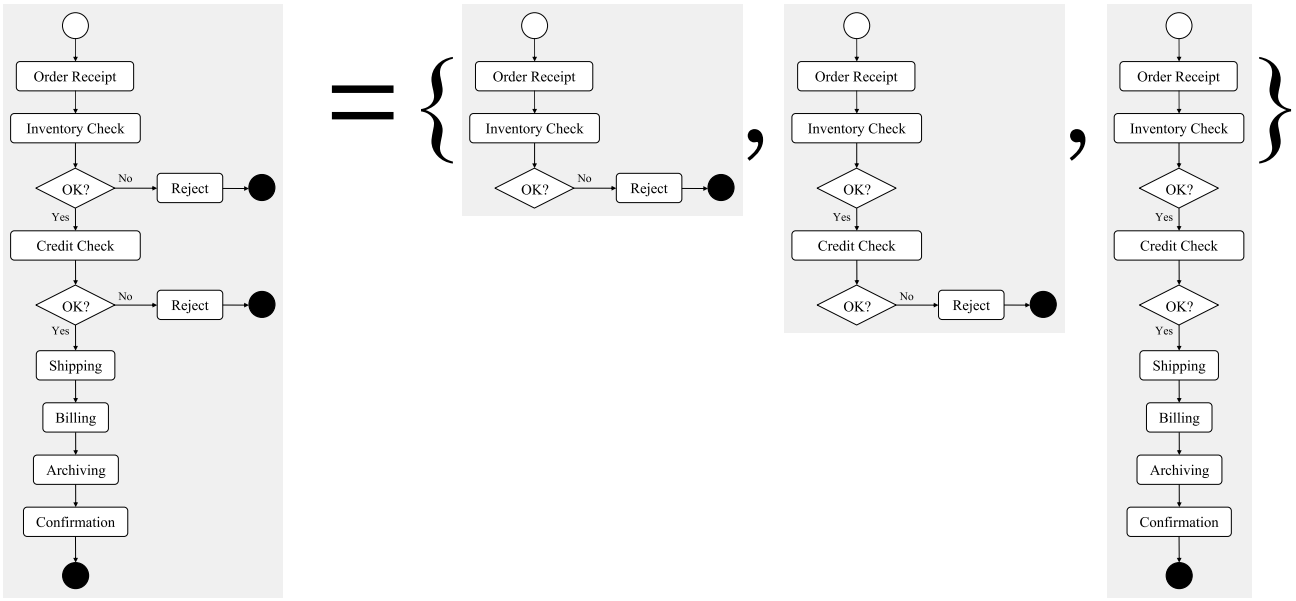


Figure 6: Expressing workflow requirements on Configuration 1 by a family of graphs (see Figure 2).

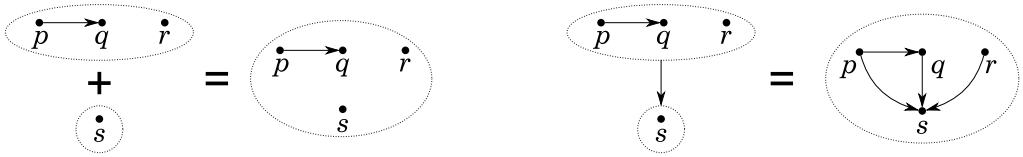


Figure 7: Parallel and sequential composition example (no common vertices).

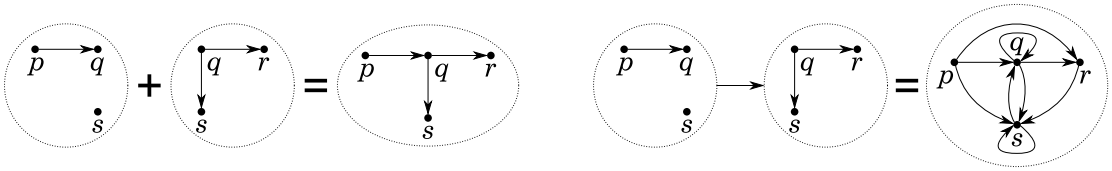


Figure 8: Parallel and sequential composition example (common vertices).

- The *parallel composition* of workflows p and q is denoted by $p + q$, e.g. Billing + Shipping as in Figure 3. The operator + is commutative, associative, and has ε as the identity:
  1) $p + q = q + p$
  2) $p + (q + r) = (p + q) + r$
  3) $p + \varepsilon = p$
- The *sequential composition* of workflows p and q is denoted by $p \to q$, e.g. Shipping → Billing as in Figure 2. The operator → has a higher precedence than +. It is associative, has the same identity ε as +, distributes over +, and can be decomposed into pairwise sequences:
  1) $p \to (q \to r) = (p \to q) \to r$
  2) $p \to \varepsilon = p$ and $\varepsilon \to p = p$
  3) $p \to (q + r) = p \to q + p \to r$ and $(p + q) \to r = p \to r + q \to r$
  4) $p \to q \to r = p \to q + p \to r + q \to r$

Figure 7 shows an example of parallel and sequential composition of graphs. It can be seen that the parallel composition does not introduce any new order dependencies between the actions coming from different graphs; therefore, they can be executed concurrently. Sequential composition, on the other hand, imposes order on the actions by introducing new dependencies between actions p, q, and r coming from the top graph and action s coming from the bottom graph. Hence, the resulting workflow behaviour is interpreted as the behaviour specified by the top graph followed by the behaviour specified by the bottom graph. Another example of these operations is shown in Figure 8. Since the graphs have common vertices, their compositions are more complicated, in particular, their sequential composi-

tion contains the self-dependencies (q, q) and (s, s) which lead to a *deadlock* in the resulting workflow. Action p can occur, but all the remaining actions are locked: neither q nor s can proceed due to the self-dependencies, while r cannot proceed without q and s. Figures 9(a) and 9(b) illustrate the distributivity and decomposition axioms, respectively.

- A *conditional workflow* is denoted by $[x]p$, where x is a predicate expressing a certain condition, e.g. 'Inventory Check OK', and p is a workflow. We postulate that $[1]p = p$ and $[0]p = \varepsilon$. This allows us to model branching in flow charts. For example, the algebraic expression

$$a \to ([x]p + [\neg x]q)$$

corresponds to a branching performed after action $a \in \mathcal{A}$, which is followed by workflow p if predicate x holds, and by workflow q if predicate x does not hold. Alternatively, one can say that the above expression corresponds to a family of graphs, in which vertex a is followed by actions from the graphs coming either from family p or from family q, as illustrated in Figure 9(c). We will use the following short-hand notation for a clearer correspondence with flow charts:

$$\begin{cases} a \xrightarrow{\text{Yes}} p \stackrel{\text{df}}{=} a \to [\text{A OK}]p \\ a \xrightarrow{\text{No}} p \stackrel{\text{df}}{=} a \to [\neg(\text{A OK})]p \end{cases}$$

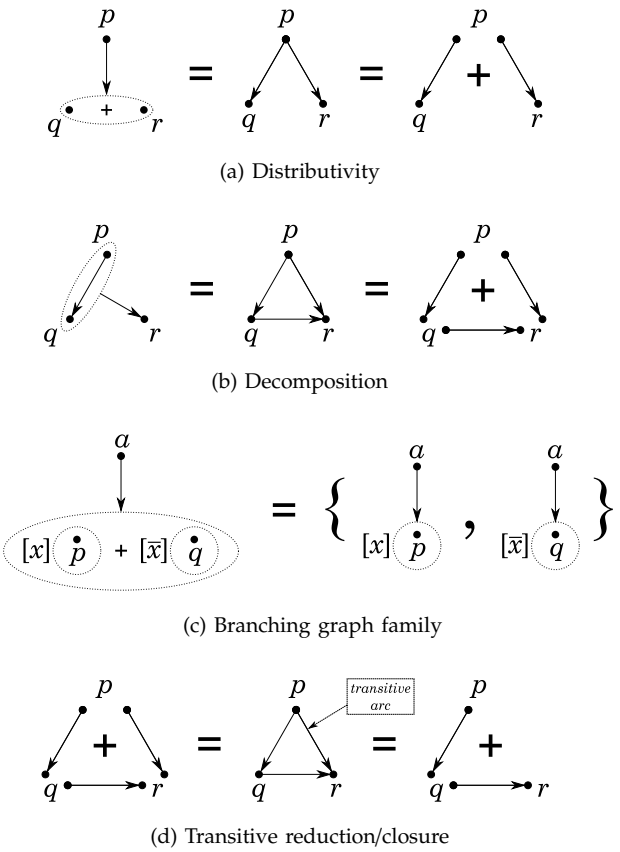
where predicate 'A OK' corresponds to the successful completion of a branching action a. For example, the expression

$$\begin{aligned} &\mathsf{InventoryCheck} \xrightarrow{\text{Yes}} \mathsf{CreditCheck} + \\ &\mathsf{InventoryCheck} \xrightarrow{\text{No}} \mathsf{Reject} \to \mathsf{End} \end{aligned}$$

corresponds to the first branching in Figure 2: if the inventory check is completed successfully the workflow continues with the credit check, otherwise the order is rejected and the workflow ends (actions Start and End denote the start/end circles that are used in the flow chart). Notice that operators $\xrightarrow{\text{Yes}}$ and $\xrightarrow{\text{No}}$ bind less tightly than $\to$ to reduce the number of parentheses. Operator $[x]$ has the highest precedence and obeys the following useful equalities:

1) $[x \wedge y]p = [x][y]p$
2) $[x \vee y]p = [x]p + [y]p$
3) $[x](p + q) = [x]p + [x]q$
4) $[x](p \to q) = [x]p \to [x]q$

To summarise, the following operators are used to create and manipulate graph families corresponding to workflows (in decreasing order of precedence):



(a) Distributivity

(b) Decomposition

(c) Branching graph family

(d) Transitive reduction/closure

Figure 9: Manipulating parameterised graphs.

$[x]p$, $p \to q$, $a \xrightarrow{\text{Yes}} p$, $a \xrightarrow{\text{No}} p$, and $p + q$, where a is an action, x is a predicate, and p and q are workflows.

The algebraic notation is used to translate flow charts into mathematical descriptions amenable to automated verification, as described in the next subsection.

B. Modelling and Analysis

We now describe how to specify workflow requirements and their reconfiguration as families of graphs, how to reason about the correctness of such specifications, and how to manipulate them using the operators of the algebra.

The algebraic notation can be used to translate the flow chart in Figure 2 into the following expression:

$$\begin{aligned} c_1 \;=\; & \mathsf{Start} \to \mathsf{OrderReceipt} \to (\\ & \quad \mathsf{InventoryCheck} \xrightarrow{\text{No}} \mathsf{Reject} \to \mathsf{End} + \\ & \quad \mathsf{InventoryCheck} \xrightarrow{\text{Yes}} (\\ & \qquad \mathsf{CreditCheck} \xrightarrow{\text{No}} \mathsf{Reject} \to \mathsf{End} + \\ & \qquad \mathsf{CreditCheck} \xrightarrow{\text{Yes}} \mathsf{Shipping} \to \mathsf{Billing} \to \\ & \qquad \mathsf{Archiving} \to \mathsf{Confirmation} \to \mathsf{End} \\ & \quad) \\ &) \end{aligned}$$

The expression can be rewritten using the axioms in order to prove that certain requirements hold. For example, any expression can be rewritten into a so-called *canonical form* [32], which is sufficient for checking most of the requirements listed in Section II-B.

Proposition 1. *Any workflow expression can be rewritten in the following canonical form* [32]:

$$\left(\sum_{a\in A}[f_a]a\right)+\left(\sum_{a,b\in A}[f_{ab}](a\to b)\right) \tag{1}$$

where:

- *A is a subset of actions that appear in the original expression;*
- *for all $a \in A$, f_a are canonical forms of Boolean expressions and are distinct from 0;*
- *for all $a, b \in A$, f_{ab} are canonical forms of Boolean expressions such that $f_{ab} \Rightarrow f_a \wedge f_b$ (that is, a dependency between actions a and b may exist only if both actions exist).*

In other words, the canonical form of an expression lists all constituent actions and all pairwise dependencies between them (along with their predicates). The canonical form can contain redundant transitive dependencies, such as $p \to r$ in presence of both $p \to q$ and $q \to r$. Such terms can be eliminated to simplify the resulting expression. This corresponds to the well-known *transitive reduction* procedure, which can be formalised by adding the following axiom [32]:

if $q \neq \varepsilon$ then $p \to q + p \to r + q \to r = p \to q + q \to r$

The axiom can be used to add or to remove transitive dependencies as necessary, see Figure 9(d).

Rewriting a workflow expression manually by following the CPOG axioms is tedious and error-prone. This motivated us to automate computation of the canonical form and the transitive reduction, as will be discussed in Section IV-D. By applying these procedures to c_1 we obtain its reduced canonical form shown below. For brevity, we will denote predicates InventoryCheck OK and CreditCheck OK simply by x and y, respectively. Actions are visually separated from dependencies by a horizontal line.

$$\begin{aligned} c_1 = {} & [1]\mathsf{Start} + [1]\mathsf{OrderReceipt} + \\ & [1]\mathsf{InventoryCheck} + [\overline{x} \vee \overline{y}]\mathsf{Reject} + \\ & [1]\mathsf{End} + [x]\mathsf{CreditCheck} + \\ & [x \wedge y]\mathsf{Billing} + [x \wedge y]\mathsf{Shipping} + \\ & [x \wedge y]\mathsf{Archiving} + [x \wedge y]\mathsf{Confirmation} + \\ \hline & [1](\mathsf{Start} \to \mathsf{OrderReceipt}) + \\ & [1](\mathsf{OrderReceipt} \to \mathsf{InventoryCheck}) + \\ & [x](\mathsf{InventoryCheck} \to \mathsf{CreditCheck}) + \\ & [\overline{x}](\mathsf{InventoryCheck} \to \mathsf{Reject}) + \\ & [x \wedge \overline{y}](\mathsf{CreditCheck} \to \mathsf{Reject}) + \\ & [\overline{x} \vee \overline{y}](\mathsf{Reject} \to \mathsf{End}) + \\ & [x \wedge y](\mathsf{CreditCheck} \to \mathsf{Shipping}) + \\ & [x \wedge y](\mathsf{Shipping} \to \mathsf{Billing}) + \\ & [x \wedge y](\mathsf{Billing} \to \mathsf{Archiving}) + \\ & [x \wedge y](\mathsf{Archiving} \to \mathsf{Confirmation}) + \\ & [x \wedge y](\mathsf{Confirmation} \to \mathsf{End}) \end{aligned}$$

The resulting expression gives us plenty of valuable information that can be used for analysis of the workflow. In particular, the following correctness properties can be verified:

- The starting and ending actions are part of the workflow regardless of possible outcomes of the branching actions, as indicated by 'unconditional' terms [1]Start and [1]End.
- The billing, shipping, archiving, and confirmation actions are performed if and only if both the inventory and internal credit checks are successful; in fact, all these actions are pre-conditioned with the predicate $x \wedge y$.
- An order is rejected if either the inventory check or the internal credit check fails, as confirmed by term $[\overline{x} \vee \overline{y}]$Reject.
- The first action after Start is always OrderReceipt, as confirmed by term [1](Start → OrderReceipt) and the lack of any other (non-transitive) dependencies on action Start.
- The shipping and billing actions are always performed sequentially: whenever the actions occur ($[x \wedge y]$Billing and $[x \wedge y]$Shipping) so does the dependency $[x \wedge y]$(Shipping → Billing).
- There are no cyclic dependencies and therefore each action is performed only once and the workflow always terminates.

We now translate the workflow requirements for Configuration 2 shown in Figure 3 into our notation, verify several relevant correctness properties, and highlight the differences between the two configurations. The flow chart in Figure 3 can be translated into the following expression:

$$
\begin{aligned}
c_2 \;=\;& \mathsf{Start} \to \mathsf{OrderReceipt} \to (\\
& \quad \mathsf{InventoryCheck} \xrightarrow{\mathrm{Yes}} cc \;+\\
& \quad \mathsf{InventoryCheck} \xrightarrow{\mathrm{No}} (\\
& \qquad \mathsf{SupplierCheck} \xrightarrow{\mathrm{Yes}} cc\\
& \qquad \mathsf{SupplierCheck} \xrightarrow{\mathrm{No}} \mathsf{Reject} \to \mathsf{End}\\
& \quad)\\
&)
\end{aligned}
$$

where expression cc corresponds to the part of the workflow starting with the CreditCheck action:

$$
\begin{aligned}
cc \;=\;& \mathsf{CreditCheck} \xrightarrow{\mathrm{No}} \mathsf{Reject} \to \mathsf{End} \;+\\
& \mathsf{CreditCheck} \xrightarrow{\mathrm{Yes}} (\mathsf{Billing} + \mathsf{Shipping}) \to\\
& \mathsf{Archiving} \to \mathsf{End}
\end{aligned}
$$

The ability to abstract and share/instantiate common behaviour (e.g. cc in the above workflow) is essential for specifying real-life reconfigurable systems, where monolithic specifications are impractical. Our prototype implementation supports abstraction and sharing (see Section IV-D for further details).

Expression c_2 can now be prepared for further analysis by converting it into the canonical form and transitively reducing the result as described above. For brevity, we denote predicates InventoryCheck OK and SupplierCheck OK by x_1 and x_2, respectively, thereby emphasising that roles of inventory and supplier checks are similar; the predicate CreditCheck OK is denoted by y as before. The resulting expression is shown below.

$$
\begin{aligned}
c_2 \;=\;& [1]\mathsf{Start} + [1]\mathsf{OrderReceipt} + [1]\mathsf{End} &+\\
& [1]\mathsf{InventoryCheck} + [\overline{x_1}]\mathsf{SupplierCheck} &+\\
& [x_1 \vee x_2]\mathsf{CreditCheck} + [\overline{x_1} \wedge \overline{x_2} \vee \overline{y}]\mathsf{Reject} &+\\
& [(x_1 \vee x_2) \wedge y]\mathsf{Billing} &+\\
& [(x_1 \vee x_2) \wedge y]\mathsf{Shipping} &+\\
& [(x_1 \vee x_2) \wedge y]\mathsf{Archiving} &+\\
\hline
& [1](\mathsf{Start} \to \mathsf{OrderReceipt}) &+\\
& [1](\mathsf{OrderReceipt} \to \mathsf{InventoryCheck}) &+\\
& [\overline{x_1}](\mathsf{InventoryCheck} \to \mathsf{SupplierCheck}) &+\\
& [x_1](\mathsf{InventoryCheck} \to \mathsf{CreditCheck}) &+\\
& [\overline{x_1} \wedge x_2](\mathsf{SupplierCheck} \to \mathsf{CreditCheck}) &+\\
& [\overline{x_1} \wedge \overline{x_2}](\mathsf{SupplierCheck} \to \mathsf{Reject}) &+\\
& [(x_1 \vee x_2) \wedge \overline{y}](\mathsf{CreditCheck} \to \mathsf{Reject}) &+\\
& [\overline{x_1} \wedge \overline{x_2} \vee \overline{y}](\mathsf{Reject} \to \mathsf{End}) &+\\
& [(x_1 \vee x_2) \wedge y](\mathsf{CreditCheck} \to \mathsf{Billing}) &+\\
& [(x_1 \vee x_2) \wedge y](\mathsf{CreditCheck} \to \mathsf{Shipping}) &+\\
& [(x_1 \vee x_2) \wedge y](\mathsf{Billing} \to \mathsf{Archiving}) &+\\
& [(x_1 \vee x_2) \wedge y](\mathsf{Shipping} \to \mathsf{Archiving}) &+\\
& [(x_1 \vee x_2) \wedge y](\mathsf{Archiving} \to \mathsf{End})
\end{aligned}
$$

Using the derived expression, the following properties of Configuration 2 can be verified:

- The billing and shipping actions are concurrent as indicated by the lack of any dependency between them. This is different from Configuration 1 where the actions could only occur in sequence: $[x \wedge y](\mathsf{Shipping} \to \mathsf{Billing})$.
- The credit check is conducted when either inventory or supplier check is successful, as indicated by term $[x_1 \vee x_2]\mathsf{CreditCheck}$. Again, this is different from Configuration 1, where there was no way around the inventory check.
- Consequently, an order is rejected under the condition $\overline{x_1} \wedge \overline{x_2} \vee \overline{y}$, that is, when either both inventory and supplier checks have failed, or the credit check has failed. By negating this condition we obtain $(x_1 \vee x_2) \wedge y$, which guards the billing, shipping, and archiving actions, as well as dependencies between them.
- The confirmation action is missing in c_2, as intended. One can also highlight this by adding a redundant term $[0]\mathsf{Confirmation}$ to c_2.

As demonstrated above, Configurations 1 and 2 have several important differences. Hence, if a system's configuration is dynamically changed from one configuration to another, the system may end up in an impossible state according to the new configuration. Such situations may lead to the system's failure, and (therefore) must be prevented. In the next subsection we discuss how the CPOG-based modelling and verification approach can be used to describe formally such situations, determine under which circumstances they can occur, and derive practicable *reconfiguration guidelines* to prevent their occurrence.

C. Dynamic reconfiguration

We start by introducing the concept of *history*, which allows reasoning about states of a system whose behaviour is described by a CPOG specification.

A *history* $H \subseteq \mathcal{A}$ is a set of actions that have occured in a system up to a certain moment in time. The set must be *causally closed*, that is, if an action has occured then all the actions it depends on must have occurred as well. Since CPOGs are capable of describing not just single workflows but families of workflows, the notion of causality becomes blurred. In fact, one can only talk about a *conditional causality*, where an action b depends on another action a under a condition x, or algebraically: $[x](a \to b)$. This leads to the following notion of consistency.

Given a history H and a CPOG specification S, we define the *consistency condition* $C(H, S)$ under which all actions in H could have occurred without violating the specification S:

$$C(H, S) = \bigwedge_{a \in H} f_a \wedge \bigwedge_{a \notin H,\ b \in H} \overline{f_{ab}}$$

where f_a and f_{ab} come from the canonical form of the specification S (see Proposition 1). In other words, a history H is consistent with a specification S if and only if:

- Every $a \in H$ must be allowed by the specification, i.e. its condition f_a must be satisfied.
- For each pair of actions a and b such that a is not in the history but b is, the dependency $a \rightarrow b$ must not be in the specification, i.e. f_{ab} must not be satisfied.

To clarify the above, consider the following example. Let $H = \{\mathsf{Start}, \mathsf{OrderReceipt}, \mathsf{InventoryCheck}, \mathsf{Reject}\}$. The consistency condition for H with respect to Configuration 1 is $C(H, c_1) = \overline{x}$, which means that history H can only be consistent if the inventory check failed, as otherwise action Reject should causally depend on CreditCheck, but the latter is not present in H. The consistency condition with respect to Configuration 2 is $C(H, c_2) = 0$, because to be rejected an order must either fail the supplier check or fail the credit check, but neither of them is in H. Therefore, we call history H *inconsistent* with the specification c_2. The reader may recognise that the notion of a history is related to (and in fact is quite similar to) the notion of a *conflict free subset of an event structure* [37].

Having defined the notion of consistency, we can now formulate the circumstances under which a system can be reconfigured from one configuration to another. If a system's state is described by a history H then it can be dynamically reconfigured from S_1 to S_2 under the condition that H is consistent with both S_1 and S_2, i.e.

$$C(H, S_1) \wedge C(H, S_2)$$

In other words, both S_1 and S_2 must be *compatible* with respect to history H. Referring to our previous example, we can say that specifications c_1 and c_2 are not compatible with respect to history $\{\mathsf{Start}, \mathsf{OrderReceipt}, \mathsf{InventoryCheck}, \mathsf{Reject}\}$.

To model unplanned reconfiguration we introduce new *reconfiguration actions* that can add and remove workflow actions and/or requirements on their order by modifying the graph family (i.e. adding new graphs and removing existing graphs that are no longer relevant). Notice that reconfiguration actions can occur concurrently with workflow actions and can also have requirements imposed on their order.

Consider a reconfiguration action r that changes the system's configuration from c_1 to c_2. The combined family of graphs which contains both c_1 and c_2, and specifies how the concurrent reconfiguration action r changes the system can be specified as

$$S = r + [\neg(r \text{ done})]c_1 + [r \text{ done}]c_2$$

where predicate 'r done' is true after reconfiguration action r has occurred and false before that.

We can now compute a set $\mathcal{R}(r, S)$ of *safe reconfiguration histories* by finding consistent histories H that remain consistent after action r occurs:

$$\mathcal{R}(r, S) = \{H \mid C(H, S) \wedge C(H \cup r, S) \neq 0\}$$

For example, we know that:

$$\{\mathsf{Start}, \mathsf{OrderReceipt}, \mathsf{InventoryCheck}, \mathsf{Reject}\} \notin \mathcal{R}(r, S)$$

However, if we drop action Reject the result is a safe reconfiguration history:

$$\{\mathsf{Start}, \mathsf{OrderReceipt}, \mathsf{InventoryCheck}\} \in \mathcal{R}(r, S)$$

An important practical question arises at this point: is it possible to derive *reconfiguration guidelines* from $\mathcal{R}(r, S)$ such that implementing the guidelines will ensure the safe reconfiguration of the system? We give a positive and constructive answer below, but with no claim for optimality.

To derive reconfiguration guidelines for the specification S defined above, notice that all histories which are consistent with c_1 and do not contain the actions Reject or Confirmation also belong to $\mathcal{R}(r, S)$:

$$\forall H, H \cap RC = \emptyset \wedge C(H, c_1) \neq 0 \Rightarrow H \in \mathcal{R}(r, S)$$

where $RC = \{\mathsf{Reject}, \mathsf{Confirmation}\}$ is a set of *forbidden actions*. Indeed, it is easy to check by examining Figures 2 and 3 that Configurations 1 and 2 are compatible with respect to any history that does not contain the forbidden actions. Therefore, we can formulate the following guideline: the reconfiguration should only be permitted when no forbidden action has occured. This guideline can be enforced by transforming the specification S into S_{safe} as follows:

$$S_{\text{safe}} = S + r \rightarrow (\mathsf{Reject} + \mathsf{Confirmation})$$

That is, we require action r to occur before the forbidden actions.

Similarly, we can ensure that the reverse reconfiguration (from c_2 to c_1) is safe by the following specification:

$$\begin{aligned} S^{\text{rev}}_{\text{safe}} \quad = \quad & r \rightarrow (\mathsf{SupplierCheck} + \mathsf{Reject} + \mathsf{Billing}) + \\ & [\neg(r \text{ done})]c_2 + [r \text{ done}]c_1 \end{aligned}$$

The reason to forbid action Billing, which seems innocuous, is that Billing must occur after Shipping in Configuration 1. If we do not forbid Billing, then a reconfiguration from c_2 to c_1 can bring the system to an inconsistency when Billing has occured but Shipping has not.

D. Implementation

We have automated CPOG transformation and verification procedures described in this section in a prototype implementation as an embedded domain-specific language (DSL) in Haskell [15]. We have also successfully cross-checked the results using Maude, a well-known environment for rewriting logic [26]. However, implementing CPOG axioms and transformations as a collection of rewrite rules in Maude was a fragile and challenging process: any new rewrite rule could trigger non-termination of the tool's rewrite engine, requring its manual shutdown and restart. Therefore, we decided to focus our further implementation effort on the DSL, where we had full control over the internal CPOG-specific rewrite engine. An additional benefit of embedding our DSL in Haskell was that the DSL acquired abstraction, sharing, and parameterisation capabilities essentially for free due to the purity and rich type system of Haskell.

As shown in [34], most interesting properties that can be defined on CPOGs are reducible to the Boolean Satisfiability (SAT) problem [8]. For example, checking the equivalence of two CPOGs requires a pairwise comparison of conditions f_a and f_{ab} of their canonical forms for equality, and each such comparison is trivially a SAT problem. When working with Maude, we relied on its built-in SAT solver, but with our Haskell-based implementation we decided to employ Binary Decision Diagrams (BDDs) [25] for storing CPOG conditions in a canonical form and to allow for sharing of their common subexpressions. We noticed that in practice actions and dependencies between them typically have the same or similar conditions; canonical forms of expressions c_1 and c_2 are good examples of this phenomenon.

E. Evaluation

We now identify the strengths and weaknesses of the CPOG formalism for modelling and verification of workflow requirements and their dynamic reconfiguration.

Strengths

The main strength of the CPOG formalism is its ability to describe workflow requirements compactly and to manipulate them in a provably correct way using the axioms defined in Section IV-A. The manipulation can be automated by reusing either standard term rewriting engines like Maude or by developing a custom proof assistant embedded in a high-level language like Haskell; we eventually followed the latter approach.

In Section IV-B, it was demonstrated that many important properties of a workflow can be expressed and efficiently verified by converting CPOG equivalence and reachability problems into SAT instances or queries to a BDD engine.

Finally, the CPOG formalism is flexible enough to express unplanned dynamic reconfiguration and formulate the relevant verification properties, as explained in Section IV-C. Importantly, both internal workflow events as well as reconfiguration events can be modelled via true concurrency semantics.

Weaknesses

There are two key weaknesses of CPOGs for modelling workflows and their reconfiguration.

First, the CPOG formalism is relatively low-level. When using CPOGs the designer is expected to operate with low-level events and conditions, and at present it is not known whether any higher level concept has a meaningful interpretation in CPOG theory.

Second is the lack of mature tool support. We were able to employ generic tools like Maude and to implement a prototype domain-specific language in Haskell. However, interoperability with other existing tool-kits is very limited at best. It is unrealistic to expect CPOGs to be used to specify a complete system. Therefore, it is essential to develop tools that enable conversion between CPOGs and well-established system design methods, such as VDM, in order to use the verification capabilites offered by CPOGs to design real-life systems.

V. Basic CCS^{dp}

Basic CCS^{dp} is a two-sorted process algebra based on basic CCS [30], which is extended with a single construct – the fraction process $\frac{P'}{P}$ – in order to describe process reconfiguration. Therefore, basic CCS^{dp} has a behavioural approach to describing a system. One sort is used to represent general purpose computation actions, the other sort is used to represent process reconfiguration actions, and interference between the two kinds of action is represented by interleaving actions. The process expressions are amenable to model checking, as we demonstrate by attempting to verify the termination requirements of the case study. We proceed by defining the syntax and semantics of basic

CCS^{dp} briefly, formulating the case study, then model checking the process expressions.

A. Syntax

Let $\mathcal{N}$ be the countable set of names (e.g. a, b, c) that represent both input ports and input actions of the processes in basic CCS^{dp}; and let $\overline{\mathcal{N}}$ be the countable set of complementary names (e.g. $\overline{a}$, $\overline{b}$, $\overline{c}$) that represent both output ports and output actions of the processes in basic CCS^{dp}, where $\overline{\mathcal{N}} \triangleq \{\overline{l} \mid l \in \mathcal{N}\}$. Let $\mathcal{PN}$ be the countable set of names (e.g. A, B, C) of the processes in basic CCS^{dp}. The sets $\mathcal{N}$, $\overline{\mathcal{N}}$, and $\mathcal{PN}$ are assumed to be pairwise disjoint.

Thus, given $a \in \mathcal{N}$, a represents the input action on the input port a of a process; and $\overline{a}$ represents the complementary output action on the output port $\overline{a}$ of a process. The interaction between complementary actions (such as a and $\overline{a}$) is represented by the special action τ, which is internal to a process.

Let $\mathcal{L}$ be the set of names that represent both ports and actions of the processes in basic CCS^{dp}, where $\mathcal{L} \triangleq \mathcal{N} \cup \overline{\mathcal{N}}$.
We define the function $\bar{} : \mathcal{L} \longrightarrow \mathcal{L}$ such that

$$\overline{x} \triangleq \begin{cases} \overline{l} & \text{if } \exists l \in \mathcal{N}(x = l) \\ l & \text{elseif } \exists \overline{l} \in \overline{\mathcal{N}}(x = \overline{l}) \end{cases}$$

so that $\forall l \in \mathcal{N}(\overline{\overline{l}} = l)$ (as required by convention).

Let $\mathcal{I}$ be the set of input and output ports/actions of the processes in basic CCS^{dp}, and their internal action (τ), where $\mathcal{I} \triangleq \mathcal{L} \cup \{\tau\}$.

Let $\mathcal{P}$ be the set of processes in basic CCS^{dp}.

The syntax of a process P in $\mathcal{P}$ is defined as follows (using the style in [39]):

$$P ::= PN\langle\widetilde{\beta}\rangle \mid M \mid P|P \mid \frac{P}{P}$$
$$M ::= 0 \mid \alpha.P \mid M + M$$

where $PN \in \mathcal{PN}$, $\widetilde{\beta}$ is a tuple of elements of $\mathcal{L}$, and $\alpha \in \mathcal{I}$.

Thus, the syntax of basic CCS^{dp} is the syntax of basic CCS without ν extended with the $\frac{P'}{P}$ construct.

As in CCS, 0 is the *NIL* process, which has no behaviour. Prefix (e.g. $\alpha.P$) models sequential action. Summation (e.g. $M + M'$) models non-deterministic choice of actions by a process. Notice that 0 can be represented as the empty summation $\sum_{\emptyset}$ (by convention). Notice also that a non-0 term in a summation is guarded by a prefix action in order to prevent the creation of an infinite number of processes, which complicates reasoning. $A\langle\widetilde{\beta}\rangle$ models the invocation of a constant process named A, instantiated with a tuple of port/action names $\widetilde{\beta}$. $A(\widetilde{\beta})$ has a unique definition, which can be recursive. Parallel composition (e.g. $P|P'$) models the execution of concurrent processes and their direct functional interaction, as well as process composition and decomposition. Interaction between processes is synchronous and point-to-point.

A *fraction* (e.g. $\frac{P'}{P}$) is a process that models process replacement and deletion. On creation, the fraction $\frac{P'}{P}$ identifies any instance of a process matching its denominator process P with which it is composed in parallel, and replaces that process atomically with the numerator process P'. If no such process instance exists, the fraction continues to exist until such a process is created (or the fraction is itself deleted or replaced). If there is more than one such process instance, a non-deterministic choice is made as to which process is replaced. Similarly, if more than one fraction can replace a process instance, a non-deterministic choice is made as to which fraction replaces the process. Deletion of a process P is achieved by parallel composition with $\frac{0}{P}$. If P progresses to Q, then $\frac{P'}{P}$ will not replace Q by P' (unless Q matches P). Notice that a fraction has no communication behaviour; its only behaviour is to replace a process with which it is composed in parallel that matches its denominator. The matching is done by behaviour using a bisimulation, as explained in the following section.

The precedence of the operators (in decreasing order) is: fraction formation, relabelling, prefix, summation, parallel composition.

B. Semantics

Let $\mathcal{R}$ be the countable set of reconfiguration actions of the processes in $\mathcal{P}$ (e.g. ρ_X, ρ_Y, ρ_Z) that create a process in $\mathcal{P}$; and let $\overline{\mathcal{R}}$ be the countable set of complementary reconfiguration actions of the processes in $\mathcal{P}$ (e.g. $\overline{\rho}_X$, $\overline{\rho}_Y$, $\overline{\rho}_Z$) that delete a process in $\mathcal{P}$, where $\overline{\mathcal{R}} \triangleq \{\overline{\rho}_X \mid \rho_X \in \mathcal{R}\}$ (see the *Creat* and *Delet* rules below). Each action in $\mathcal{R}$ is represented by ρ_X, with $X \in \mathcal{P}$. The sets $\mathcal{N}$, $\overline{\mathcal{N}}$, $\{\tau\}$, $\mathcal{R}$, $\overline{\mathcal{R}}$, and $\mathcal{PN}$ are assumed to be pairwise disjoint[2].

The interaction between complementary reconfiguration actions (such as ρ_X and $\overline{\rho}_X$) results in the replacement of one or more processes (see the *Creat*, *Delet*, and *React* rules below).

Let $\mathcal{C}$ be the set of reconfiguration actions of the processes in $\mathcal{P}$, where $\mathcal{C} \triangleq \mathcal{R} \cup \overline{\mathcal{R}}$.

[2]The reconfiguration actions ρ_X, $\overline{\rho}_X$ are written as τ_{r_X}, $\overline{\tau}_{r_X}$ respectively in [3]. The change to ρ_X, $\overline{\rho}_X$ simplifies the notation, and is due to advice from Honda.

We extend the definition of the function $\bar{}$ as follows:
$\bar{} : \mathcal{L} \cup \mathcal{C} \longrightarrow \mathcal{L} \cup \mathcal{C}$ such that

$$\overline{\lambda} \triangleq \begin{cases} \bar{l} & \text{if } \exists l \in \mathcal{N}(\lambda = l) \\ l & \text{elseif } \exists \bar{l} \in \overline{\mathcal{N}}(\lambda = \bar{l}) \\ \overline{\rho}_X & \text{elseif } \exists \rho_X \in \mathcal{R}(\lambda = \rho_X) \\ \rho_X & \text{elseif } \exists \overline{\rho}_X \in \overline{\mathcal{R}}(\lambda = \overline{\rho}_X) \end{cases}$$

so that $\forall \lambda \in \mathcal{L} \cup \mathcal{C}\, (\overline{\overline{\lambda}} = \lambda)$ (as required by convention).

Let $\mathcal{A}$ be the set of actions of the processes in $\mathcal{P}$, where $\mathcal{A} \triangleq \mathcal{I} \cup \mathcal{C}$.

The labelled transition system (LTS) rules for basic CCS^{dp} are a superset of the LTS rules for basic CCS without ν, consisting of an unchanged rule of basic CCS (i.e. *Sum*) plus basic CCS rules applicable to reconfiguration transitions (i.e. *React*, *L-Par*, *R-Par*, and *Ident*) plus additional rules to describe new reconfiguration behaviour (i.e. *Creat*, *Delet*, *CompDelet*, *L-React*, and *R-React*). See Table I.

Sum	$\frac{k \in I}{\sum_{i \in I} \alpha_i.P_i \xrightarrow{\alpha_k} P_k}$ where I is a finite indexing set
React	$\frac{\lambda \in \mathcal{L} \cup \mathcal{C} \ \wedge \ P \xrightarrow{\lambda} P' \ \wedge \ Q \xrightarrow{\overline{\lambda}} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}$
L-Par	$\frac{\mu \in \mathcal{A} \ \wedge \ P \xrightarrow{\mu} P'}{P \mid Q \xrightarrow{\mu} P' \mid Q}$
R-Par	$\frac{\mu \in \mathcal{A} \ \wedge \ Q \xrightarrow{\mu} Q'}{P \mid Q \xrightarrow{\mu} P \mid Q'}$
Ident	$\frac{\lvert\widetilde{b}\rvert = \lvert\widetilde{a}\rvert \ \wedge \ \mu \in \mathcal{A} \ \wedge \ P[\frac{\widetilde{b}}{\widetilde{a}}] \xrightarrow{\mu} P'}{A<\widetilde{b}> \xrightarrow{\mu} P'}$ where $A(\widetilde{a}) \triangleq P$
Creat	$\frac{P \sim_{of} Q \ \wedge \ P \in \mathcal{P}^+}{\frac{P'}{P} \xrightarrow{\rho_Q} P'}$
Delet	$\frac{P \sim_{of} Q \ \wedge \ P \in \mathcal{P}^+}{P \xrightarrow{\overline{\rho}_Q} 0}$
CompDelet	$\frac{R \sim_{of} R_1 \mid R_2 \ \wedge \ P \xrightarrow{\overline{\rho}_{R_1}} P' \ \wedge \ P' \xrightarrow{\overline{\rho}_{R_2}} P''}{P \xrightarrow{\overline{\rho}_R} P''}$
L-React	$\frac{R \sim_{of} R_1 \mid R_2 \ \wedge \ P \xrightarrow{\overline{\rho}_{R_1}} P' \ \wedge \ P' \xrightarrow{\rho_R} P'' \ \wedge \ Q \xrightarrow{\overline{\rho}_{R_2}} Q'}{P \mid Q \xrightarrow{\tau} P'' \mid Q'}$
R-React	$\frac{R \sim_{of} R_1 \mid R_2 \ \wedge \ P \xrightarrow{\overline{\rho}_{R_1}} P' \ \wedge \ Q \xrightarrow{\overline{\rho}_{R_2}} Q' \ \wedge \ Q' \xrightarrow{\rho_R} Q''}{P \mid Q \xrightarrow{\tau} P' \mid Q''}$

Table I: Labelled transition system semantics of basic CCS^{dp}.

$$\text{Sum} \ : \ \frac{k \in I}{\sum_{i \in I} \alpha_i.P_i \xrightarrow{\alpha_k} P_k} \ \text{where } I \text{ is a finite indexing set}$$

The *Sum* rule states that summation preserves the transitions of constituent processes as a non-deterministic choice of alternative transitions.

$$\text{React} \ : \ \frac{\lambda \in \mathcal{L} \cup \mathcal{C} \ \wedge \ P \xrightarrow{\lambda} P' \ \wedge \ Q \xrightarrow{\overline{\lambda}} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}$$

The *React* rule states that if two processes can perform complementary transitions, then their parallel composition can result in a τ transition in which both processes undergo their respective complementary transitions atomically.

$$\text{L-Par} \ : \ \frac{\mu \in \mathcal{A} \ \wedge \ P \xrightarrow{\mu} P'}{P \mid Q \xrightarrow{\mu} P' \mid Q}$$

$$\text{R-Par} \ : \ \frac{\mu \in \mathcal{A} \ \wedge \ Q \xrightarrow{\mu} Q'}{P \mid Q \xrightarrow{\mu} P \mid Q'}$$

The *L-Par* and *R-Par* rules state that parallel composition preserves the transitions of constituent processes.

$$\text{Ident} \ : \ \frac{\lvert\widetilde{b}\rvert = \lvert\widetilde{a}\rvert \ \wedge \ \mu \in \mathcal{A} \ \wedge \ P[\frac{\widetilde{b}}{\widetilde{a}}] \xrightarrow{\mu} P'}{A<\widetilde{b}> \xrightarrow{\mu} P'} \ \text{where } A(\widetilde{a}) \triangleq P$$

The *Ident* rule states that if a relabelled constant process can perform a given transition, then the constant process instantiated with the new labelling can also undergo the same transition.

Let $\mathcal{P}^+$ be the set of *positive processes* of $\mathcal{P}$, where $\mathcal{P}^+$ is defined to be the smallest subset of $\mathcal{P}$ that satisties the following conditions:

1. $\forall \alpha \in \mathcal{I} \ \forall p \in \mathcal{P} \ (\alpha.p \in \mathcal{P}^+)$
2. $\forall p, q \in \mathcal{P} \ (p + q \in \mathcal{P} \wedge (p \in \mathcal{P}^+ \vee q \in \mathcal{P}^+) \Longrightarrow p + q \in \mathcal{P}^+)$
3. $\forall p, q \in \mathcal{P} \ (p \in \mathcal{P}^+ \vee q \in \mathcal{P}^+ \Longrightarrow p \mid q \in \mathcal{P}^+)$
4. $\forall p \in \mathcal{P} \ \forall q \in \mathcal{P}^+ \left(\frac{p}{q} \in \mathcal{P}^+\right)$
5. $\forall \beta \in \mathcal{I} \ \forall X \in \mathcal{P}\mathcal{N} \ (\beta.X \in \mathcal{P}^+)$

Strong of-bisimulation ($\sim_{of}$) is the largest symmetric binary relation on $\mathcal{P}$ such that the following condition holds $\forall (p, q) \in \sim_{of}$
$\forall \alpha \in \mathcal{I}_p \cup \mathcal{R}_p \ \forall p' \in \mathcal{P}$
$(p \xrightarrow{\alpha} p' \Longrightarrow \alpha \in \mathcal{I}_q \cup \mathcal{R}_q \wedge \exists q' \in \mathcal{P} \ (q \xrightarrow{\alpha} q' \wedge (p', q') \in \sim_{of}))$

$$\text{Creat} \ : \ \frac{P \sim_{of} Q \ \wedge \ P \in \mathcal{P}^+}{\frac{P'}{P} \xrightarrow{\rho_Q} P'}$$

$$\text{Delet} \ : \ \frac{P \sim_{of} Q \ \wedge \ P \in \mathcal{P}^+}{P \xrightarrow{\overline{\rho}_Q} 0}$$

The *Creat* rule states that if P is a positive process ($P \in \mathcal{P}^+$) that matches Q using strong of-bisimulation ($P \sim_{of} Q$), then the fraction process $\frac{P'}{P}$ can perform

the reconfiguration transition ρ_Q that results in the creation of P'. The *Delet* rule is complementary to the *Creat* rule. It states that if P is a positive process that matches Q using strong of-bisimulation, then P can be deleted by performing the reconfiguration transition $\overline{\rho}_Q$ that is complementary to the reconfiguration transition ρ_Q performed by some fraction that creates a process. Thus, P' replaces P as a result of the reaction between $\frac{P'}{P}$ performing ρ_Q and P performing $\overline{\rho}_Q$ (defined by the *React* rule).

The hypotheses of *Creat* and *Delet* restrict reconfiguration transitions to positive processes in order to retain the identity property of 0 in parallel compositions up to $\sim_{of}$.

Strong of-bisimulation is used for process matching for two reasons. First, strong of-bisimulation is a relation between processes. The use of a relation avoids modelling reconfiguration mechanisms and operators, which (respectively) simplifies models and facilitates the abstract modelling of unplanned reconfiguration. The use of reconfiguration transitions in the LTS semantics, and not in the syntax, also helps to achieve these objectives. Thus, the relation is a pre-condition that allows a process to be reconfigured only when it is in a specified state, which is an important requirement for reconfigurable systems. Furthermore, the separation of fractions and mechanisms enables a fraction process to be combined with general purpose triggering mechanisms (such as prefixes, interrupts, and timeouts) without changing their semantics. Second, strong of-bisimulation helps to maximize the terseness of expressions modelling reconfiguration, which simplifies modelling.

Notice that reconfiguration transitions do not involve any communication. Therefore, the interaction between complementary reconfiguration transitions does not require a port or a communication channel.

The mutual dependency between LTS transitions and strong of-bisimulation suggests the dependency is circular, which is problematic. However, the dependency is an inductive relationship if the depth of fractional recursion of a process is bounded suitably. Therefore, we restrict $\mathcal{P}$ to the domain of the *sfdrdepth* function (defined below), which creates an inductive relationship between LTS transitions and strong of-bisimulation, and thereby avoids a circular dependency. *sfdrdepth* is defined as follows:

$$succ : \mathcal{P} \times \mathbb{N} \longrightarrow \mathbb{P}\,\mathcal{P} \text{ such that}$$
$$succ(p,i) \triangleq \begin{cases} \{p\} & \text{if } i = 0 \\ \{q' \in \mathcal{P} \mid \exists q \in succ(p,i-1)(\exists \alpha \in \mathcal{I}_q \cup \mathcal{R}_q (q \xrightarrow{\alpha} q'))\} & \text{else} \end{cases}$$

$succ(p,i)$ is the set of i^{th} *successor processes* (or equivalently, i^{th} *successors*) of p. That is, the set of processes reached after i consecutive transitions in $\mathcal{I} \cup \mathcal{R}$ starting from p, with $succ(p,0) = \{p\}$.

$$successors : \mathcal{P} \longrightarrow \mathbb{P}\,\mathcal{P} \text{ such that}$$
$$successors(p) \triangleq \bigcup_{i \in \mathbb{N}} succ(p,i)$$

$successors(p)$ is the set of all the successors of p, including p. That is, the set of all the processes reached after zero, one or more consecutive transitions in $\mathcal{I} \cup \mathcal{R}$ starting from p.

$$sfdrdepth : \mathcal{P} \longrightarrow \mathbb{N} \text{ such that}$$
$$sfdrdepth(p) \triangleq max\{fdrdepth(s) \mid s \in successors(p)\} \text{ with}$$

$$fdrdepth : \mathcal{P} \longrightarrow \mathbb{N} \text{ such that}$$
$$fdrdepth(s) \triangleq \begin{cases} 0 & \text{if } \mathcal{R}_s = \emptyset \\ 1 + max\{sfdrdepth(X) \mid \rho_X \in \mathcal{R}_s\} & \text{else} \end{cases}$$

$$\text{CompDelet} : \frac{R \sim_{of} R_1|R_2 \ \wedge \ P \xrightarrow{\overline{\rho}_{R_1}} P' \ \wedge \ P' \xrightarrow{\overline{\rho}_{R_2}} P''}{P \xrightarrow{\overline{\rho}_R} P''}$$

The *CompDelet* rule states that consecutive delete transitions of a process can be composed into a single delete transition of the process. The rule is applicable only if it is used in combination with *L-Par* or *R-Par*.

$$\text{L-React} : \frac{R \sim_{of} R_1|R_2 \ \wedge \ P \xrightarrow{\overline{\rho}_{R_1}} P' \ \wedge \ P' \xrightarrow{\rho_R} P'' \ \wedge \ Q \xrightarrow{\overline{\rho}_{R_2}} Q'}{P|Q \xrightarrow{\tau} P''|Q'}$$

$$\text{R-React} : \frac{R \sim_{of} R_1|R_2 \ \wedge \ P \xrightarrow{\overline{\rho}_{R_1}} P' \ \wedge \ Q \xrightarrow{\overline{\rho}_{R_2}} Q' \ \wedge \ Q' \xrightarrow{\rho_R} Q''}{P|Q \xrightarrow{\tau} P'|Q''}$$

In a process expression, the denominator of a fraction process can match the parallel composition of processes that are located on different sides of the fraction. The *L-React* and *R-React* rules state that a reconfiguration reaction can occur in this case, with all the processes participating in the reaction undergoing their respective transitions atomically.

In CCS, the parallel composition operator is associative and commutative with respect to strong bisimulation, and it is desirable to retain these properties of parallel composition with respect to strong of-bisimulation in basic CCS^{dp}, because associativity and commutativity support equational reasoning and the abstract modelling of unplanned reconfiguration. However, the denominator of a fraction can match the parallel composition of two or more processes, which enables the fraction to replace multiple processes

atomically. The replacement of these processes must be possible even if the processes are parenthesized differently or are reordered, in order to preserve the associativity and commutativity of parallel composition with respect to strong of-bisimulation. The *CompDelet* rule helps to ensure associativity, and the *L-React* and *R-React* rules help to ensure commutativity, of parallel composition with respect to strong of-bisimulation.

The LTS transitions are defined to be the smallest relation on $\mathcal{P}$ that satisfies the LTS rules. Therefore, a process $p \in \mathcal{P}$ performs a transition $p \xrightarrow{\mu} p'$ with $\mu \in \mathcal{A}$ and $p' \in \mathcal{P}$ if and only if the hypothesis of some LTS rule that determines the $p \xrightarrow{\mu} p'$ transition is satisfied.

C. Modelling

We now formulate Design 3 of both configurations of the workflow and Design 3 of the reconfiguration in basic CCS^{dp}. The sets of possible customer identifiers, product identifiers and order identifiers is assumed to be finite.

Let C be the set of possible customer identifiers,
let I be the set of possible product identifiers,
let O be the set of possible order identifiers,
such that $|C|, |I|, |O| \in \mathbb{N}^+$

Modelling Configuration 1

Configuration 1 consists of a collection of activities. Each activity consists of a collection of actions, which are modelled as actions of processes in $\mathcal{P}$. For example, the actions of the `Order Receipt` activity are modelled as the actions of the *REC* process. The actions of the `Evaluation` activity are modelled as actions of the processes *IC*, *ICH*, *CC*, and *CCH*. The `Rejection` activity is modelled as the action $\overline{Reject}_{c,i,o}$ in the processes *ICH* and *CCH*. The `Confirmation` activity is modelled as the action $\overline{Confirm}_{c,i,o}$ in the *ARCH* process.

Configuration 1 of the workflow is denoted by the process *WORKFLOW*, and
$WORKFLOW \triangleq$
$REC \mid IC \mid ICH \mid CC \mid CCH \mid SHIP \mid BILL \mid ARC \mid ARCH$

$REC \triangleq$
$\sum_{c\in C, i\in I, o\in O} Receipt_{c,i,o}.(WORKFLOW \mid \overline{InventoryCheck}_{c,i,o})$
and denotes the `Order Receipt` activity.

Notice that the subscripted actions (such as $Receipt_{c,i,o}$) are distinct. Thus,
$Receipt_{c,i,o} = Receipt_{c',i',o'} \iff c = c' \wedge i = i' \wedge o = o'$
By convention, we omit the 0 process at the end of a trace of actions by a process.

$IC \triangleq \sum_{c\in C, i\in I, o\in O} InventoryCheck_{c,i,o}.\tau.$
$(\overline{InventoryCheckNotOK}_{c,i,o} + \overline{InventoryCheckOK}_{c,i,o})$
and denotes the `Inventory Check` action in `Evaluation`.

$ICH \triangleq \sum_{c\in C, i\in I, o\in O} InventoryCheckNotOK_{c,i,o}.\overline{Reject}_{c,i,o} +$
$InventoryCheckOK_{c,i,o}.\overline{CreditCheck}_{c,i,o}$
and denotes actions in `Evaluation` and `Rejection`.

$CC \triangleq \sum_{c\in C, i\in I, o\in O} CreditCheck_{c,i,o}.\tau.$
$(\overline{CreditCheckNotOK}_{c,i,o} + \overline{CreditCheckOK}_{c,i,o})$
and denotes the `Credit Check` action in `Evaluation`.

$CCH \triangleq \sum_{c\in C, i\in I, o\in O} CreditCheckNotOK_{c,i,o}.\overline{Reject}_{c,i,o} +$
$CreditCheckOK_{c,i,o}.\overline{Ship}_{c,i,o}$
and denotes actions in `Evaluation` and `Rejection`.

$SHIP \triangleq \sum_{c\in C, i\in I, o\in O} Ship_{c,i,o}.\tau.\overline{Bill}_{c,i,o}$
and denotes the `Shipping` activity.

$BILL \triangleq \sum_{c\in C, i\in I, o\in O} Bill_{c,i,o}.\tau.\overline{Archive}_{c,i,o}$
and denotes the `Billing` activity.

$ARC \triangleq \sum_{c\in C, i\in I, o\in O} Archive_{c,i,o}.\tau.\overline{ArchiveOK}_{c,i,o}$
and denotes the `Archiving` activity.

$ARCH \triangleq \sum_{c\in C, i\in I, o\in O} ArchiveOK_{c,i,o}.\overline{Confirm}_{c,i,o}$
and denotes the `Confirmation` activity.

$\therefore WORKFLOW =$
$\sum_{c\in C, i\in I, o\in O} Receipt_{c,i,o}.(WORKFLOW \mid \overline{InventoryCheck}_{c,i,o}) \mid$
$\sum_{c\in C, i\in I, o\in O} InventoryCheck_{c,i,o}.\tau.$
$(\overline{InventoryCheckNotOK}_{c,i,o} + \overline{InventoryCheckOK}_{c,i,o}) \mid$
$\sum_{c\in C, i\in I, o\in O} InventoryCheckNotOK_{c,i,o}.\overline{Reject}_{c,i,o} +$
$InventoryCheckOK_{c,i,o}.\overline{CreditCheck}_{c,i,o} \mid$
$\sum_{c\in C, i\in I, o\in O} CreditCheck_{c,i,o}.\tau.$
$(\overline{CreditCheckNotOK}_{c,i,o} + \overline{CreditCheckOK}_{c,i,o}) \mid$
$\sum_{c\in C, i\in I, o\in O} CreditCheckNotOK_{c,i,o}.\overline{Reject}_{c,i,o} +$
$CreditCheckOK_{c,i,o}.\overline{Ship}_{c,i,o} \mid$
$\sum_{c\in C, i\in I, o\in O} Ship_{c,i,o}.\tau.\overline{Bill}_{c,i,o} \mid$
$\sum_{c\in C, i\in I, o\in O} Bill_{c,i,o}.\tau.\overline{Archive}_{c,i,o} \mid$
$\sum_{c\in C, i\in I, o\in O} Archive_{c,i,o}.\tau.\overline{ArchiveOK}_{c,i,o} \mid$
$\sum_{c\in C, i\in I, o\in O} ArchiveOK_{c,i,o}.\overline{Confirm}_{c,i,o}$

The execution of Configuration 1 of the workflow is modelled as transitions of the *WORKFLOW* process. If a $\overline{Reject}_{c,i,o}$ action is performed in *ICH* or in *CCH*, the processes to be executed subsequently in *WORKFLOW* are deleted implicitly (i.e. garbage collected). The process deletion can be represented explicitly (e.g. as $\overline{Reject}_{c,i,o}.\overline{0}$ in $\overline{CC \mid CCH \mid SHIP \mid BILL \mid ARC \mid ARCH}$ in *ICH*), but this has the disadvantage of encoding information about the workflow's structure within a workflow process, which (in general) complicates reconfiguration of workflows.

Modelling Configuration 2

Configuration 2 is different in structure from Configuration 1, although some of the activities are unchanged (such as `Inventory Check` and `Credit Check`), and this difference is reflected in the processes used to model Configuration 2. For example, the *REC* process must be different in order to spawn a workflow with Configuration 2. A new process *SC* is needed in order to model the new action (`Supplier Check`) in the `Evaluation` activity. The *CCH* process must be different in order to ensure that `Shipping` and `Billing` are performed concurrently. The removal of the `Confirmation` activity implies the `Archiving` activity no longer produces an output, and (therefore) the *ARC* process must be different and the *ARCH* process is no longer needed.

Configuration 2 of the workflow is denoted by the process *WORKFLOW'*, and
$WORKFLOW' \triangleq$
$REC' \mid IC \mid ICH' \mid CC \mid CCH' \mid SHIP' \mid BILL' \mid ARC'$

$REC' \triangleq$
$\sum_{c\in C,i\in I,o\in O} Receipt_{c,i,o}.(WORKFLOW' \mid \overline{InventoryCheck}_{c,i,o})$
and denotes the changed `Order Receipt` activity that now spawns a workflow with Configuration 2.

$ICH' \triangleq$
$\sum_{c\in C,i\in I,o\in O} InventoryCheckNotOK_{c,i,o}.(\overline{SupplierCheck}_{c,i,o} \mid SC) +$
$InventoryCheckOK_{c,i,o}.\overline{CreditCheck}_{c,i,o}$
and denotes actions in `Evaluation` that initiate a `Supplier Check` or a `Credit Check`.

$SC \triangleq \sum_{c\in C,i\in I,o\in O} SupplierCheckNotOK_{c,i,o}.\overline{Reject}_{c,i,o} +$
$SupplierCheckOK_{c,i,o}.\overline{CreditCheck}_{c,i,o}$
and denotes the new `Supplier Check` handling action in `Evaluation` followed by either `Reject` or initiation of a `Credit Check`.

$CCH' \triangleq \sum_{c\in C,i\in I,o\in O} CreditCheckNotOK_{c,i,o}.\overline{Reject}_{c,i,o} +$
$CreditCheckOK_{c,i,o}.(\overline{Ship}_{c,i,o} \mid \overline{Bill}_{c,i,o})$
and denotes actions in `Evaluation` and `Rejection`, including the concurrent initiation of `Shipping` and `Billing`.

$SHIP' \triangleq \sum_{c\in C,i\in I,o\in O} Ship_{c,i,o}.\tau.\overline{ShipOK}_{c,i,o}$
and denotes the changed `Shipping` activity that allows `Shipping` and `Billing` to be performed concurrently.

$BILL' \triangleq \sum_{c\in C,i\in I,o\in O} Bill_{c,i,o}.\tau.\overline{BillOK}_{c,i,o}$
and denotes the changed `Billing` activity that allows `Shipping` and `Billing` to be performed concurrently.

$ARC' \triangleq \sum_{c\in C,i\in I,o\in O} ShipOK_{c,i,o}.BillOK_{c,i,o}.\tau +$
$BillOK_{c,i,o}.ShipOK_{c,i,o}.\tau$
and denotes the changed `Archiving` activity that now terminates the workflow.

$\therefore WORKFLOW' =$
$\sum_{c\in C,i\in I,o\in O} Receipt_{c,i,o}.(WORKFLOW' \mid \overline{InventoryCheck}_{c,i,o}) \mid$
$\sum_{c\in C,i\in I,o\in O} InventoryCheck_{c,i,o}.\tau.$
$(InventoryCheckNotOK_{c,i,o} + \overline{InventoryCheckOK}_{c,i,o}) \mid$
$\sum_{c\in C,i\in I,o\in O} InventoryCheckNotOK_{c,i,o}.(\overline{SupplierCheck}_{c,i,o} \mid SC) +$
$InventoryCheckOK_{c,i,o}.\overline{CreditCheck}_{c,i,o} \mid$
$\sum_{c\in C,i\in I,o\in O} CreditCheck_{c,i,o}.\tau.$
$(\overline{CreditCheckNotOK}_{c,i,o} + \overline{CreditCheckOK}_{c,i,o}) \mid$
$\sum_{c\in C,i\in I,o\in O} CreditCheckNotOK_{c,i,o}.\overline{Reject}_{c,i,o} +$
$CreditCheckOK_{c,i,o}.(\overline{Ship}_{c,i,o} \mid \overline{Bill}_{c,i,o}) \mid$
$\sum_{c\in C,i\in I,o\in O} Ship_{c,i,o}.\tau.\overline{ShipOK}_{c,i,o} \mid$
$\sum_{c\in C,i\in I,o\in O} Bill_{c,i,o}.\tau.\overline{BillOK}_{c,i,o} \mid$
$\sum_{c\in C,i\in I,o\in O} ShipOK_{c,i,o}.BillOK_{c,i,o}.\tau + BillOK_{c,i,o}.ShipOK_{c,i,o}.\tau$

The execution of Configuration 2 of the workflow is modelled as transitions of the *WORKFLOW'* process. Processes are deleted implicitly following the execution of a $\overline{Reject}_{c,i,o}$ action (as in *WORKFLOW*).

Modelling the Reconfiguration

The workflow is reconfigured by a reconfiguration manager (modelled by the process *RM*) that is activated after receiving a triggering message and reconfigures the workflow from Configuration 1 to Configuration 2. There are three different ways of reconfiguring the workflow (depending on its state of execution), and they are triggered by different messages. The *trigger*1 guard models receipt of the message that is used to trigger reconfiguration of the workflow if it has **not** yet started to execute. After the release of *trigger*1, *RM* replaces the process *WORKFLOW* with the process *WORKFLOW'*, and replicates itself. The *trigger*2 guard models receipt of the message that is used to trigger reconfiguration of the workflow if it has completed `Order Receipt` and `Inventory Check` but not yet determined the action resulting from the `Inventory Check`. After the release of *trigger*2, *RM* deletes the process *ARCH*, replaces the processes *ICH*, *CCH*, *SHIP*, *BILL*, and *ARC* with the processes *ICH'*, *CCH'*, *SHIP'*, *BILL'*, and *ARC'* (respectively), and replicates itself. The *trigger*3 guard models receipt of the message that is used to trigger reconfiguration of the workflow if it has completed `Order Receipt`, `Inventory Check` (with positive result), and `Credit Check` but not yet determined the action resulting from the `Credit Check`. After the release of *trigger*3, *RM* deletes the process *ARCH*, replaces the processes *CCH*, *SHIP*, *BILL*, and *ARC* with the processes *CCH'*, *SHIP'*, *BILL'*, and *ARC'* (respectively), and replicates itself.

The reconfiguration manager is denoted by the process *RM*, and

$$RM \triangleq trigger1.\left(\frac{WORKFLOW'}{WORKFLOW} \mid RM\right) + trigger2.\left(\frac{ICH'}{ICH} \mid \frac{CCH'}{CCH} \mid \frac{SHIP'}{SHIP} \mid \frac{BILL'}{BILL} \mid \frac{ARC'}{ARC} \mid \frac{0}{ARCH} \mid RM\right) + trigger3.\left(\frac{CCH'}{CCH} \mid \frac{SHIP'}{SHIP} \mid \frac{BILL'}{BILL} \mid \frac{ARC'}{ARC} \mid \frac{0}{ARCH} \mid RM\right)$$

Thus, *RM* performs two operations of unplanned process reconfiguration, namely, the deletion and replacement of processes that are not designed to be reconfigured.

The reconfiguration of the workflow is expressed as reactions between *WORKFLOW* and *RM* in the expression *WORKFLOW* | *RM*. The step through which the reconfiguring process *RM* is added to the context of the *WORKFLOW* process, that is, the step through which *WORKFLOW* becomes *WORKFLOW* | *RM*, is performed outside basic CCS^{dp}, and thereby captures the fact that the reconfiguration is unplanned. Notice that because *RM* is located in the context of *WORKFLOW* and contains *WORKFLOW'*, Configuration 2 is located in the environment of the workflow (i.e. Configuration 2 is **not** pre-defined within the workflow). Thus, an arbitrary number of configurations and reconfigurations can be represented for the workflow, which can be used to represent its dynamic evolution.

The concurrent execution of the workflow and reconfiguration activities is represented as the set of total orders (i.e. sequences) of the transitions of the processes modelling the activities, and the functional interference between the activities is represented as interleaved transitions of the processes, as shown below.

D. Analysis

The process expressions *WORKFLOW*, *WORKFLOW'*, and *WORKFLOW* | *RM* can be used to verify by model checking whether or not their respective termination requirements hold. Verifying termination is also useful for verifying the other requirements.

We represent the termination of a workflow as the termination of the process instance representing the workflow. Therefore, a workflow with Configuration 1 terminates if the instance of *WORKFLOW* representing the workflow always reaches 0. We show the elaboration of *WORKFLOW* always reaches 0:

$$\begin{aligned}
&WORKFLOW \\
&\xrightarrow{Receipt_{c,i,o}} WORKFLOW \mid \overline{InventoryCheck}_{c,i,o} \mid IC \mid \\
&\quad ICH \mid CC \mid CCH \mid SHIP \mid BILL \mid ARC \mid ARCH \\
&\xrightarrow{\tau} WORKFLOW \mid \\
&\quad \tau.(\overline{InventoryCheckNotOK}_{c,i,o} + \overline{InventoryCheckOK}_{c,i,o}) \mid \\
&\quad ICH \mid CC \mid CCH \mid SHIP \mid BILL \mid ARC \mid ARCH
\end{aligned}$$

Thus, Requirement B.1 on Configuration 1 is satisfied, because there are a finite number of *WORKFLOW* processes, and the different instances of *WORKFLOW* do not interact with each other.

$$\begin{aligned}
&\xrightarrow{\tau} WORKFLOW \mid \\
&\quad \overline{InventoryCheckNotOK}_{c,i,o} + \overline{InventoryCheckOK}_{c,i,o} \mid \\
&\quad ICH \mid CC \mid CCH \mid SHIP \mid BILL \mid ARC \mid ARCH
\end{aligned}$$

Thus, Requirement B.2 on Configuration 1 is satisfied. For the `Inventory Check` of order *o*, $\overline{InventoryCheckNotOK}_{c,i,o}$ represents the negative outcome, $\overline{InventoryCheckOK}_{c,i,o}$ represents the positive outcome, and $\overline{InventoryCheckNotOK}_{c,i,o} + \overline{InventoryCheckOK}_{c,i,o}$ represents the non-deterministic choice between the two outcomes. The negative outcome of `Inventory Check` results in the following elaboration:

$$\begin{aligned}
&\xrightarrow{\tau} WORKFLOW \mid \overline{Reject}_{c,i,o} \\
&\quad CC \mid CCH \mid SHIP \mid BILL \mid ARC \mid ARCH \\
&\xrightarrow{\overline{Reject}_{c,i,o}} WORKFLOW \mid 0 \\
&\quad CC \mid CCH \mid SHIP \mid BILL \mid ARC \mid ARCH
\end{aligned}$$

Thus, Requirement B.3 on Configuration 1 is satisfied, the workflow terminates, and the processes *CC*, *CCH*, *SHIP*, *BILL*, *ARC*, and *ARCH* are deleted implicitly. The positive outcome of `Inventory Check` results in the following elaboration:

$$\begin{aligned}
&\xrightarrow{\tau} WORKFLOW \mid \overline{CreditCheck}_{c,i,o} \mid CC \mid \\
&\quad CCH \mid SHIP \mid BILL \mid ARC \mid ARCH \\
&\xrightarrow{\tau} WORKFLOW \mid \\
&\quad \tau.(\overline{CreditCheckNotOK}_{c,i,o} + \overline{CreditCheckOK}_{c,i,o}) \mid \\
&\quad CCH \mid SHIP \mid BILL \mid ARC \mid ARCH \\
&\xrightarrow{\tau} WORKFLOW \mid \\
&\quad \overline{CreditCheckNotOK}_{c,i,o} + \overline{CreditCheckOK}_{c,i,o} \mid \\
&\quad CCH \mid SHIP \mid BILL \mid ARC \mid ARCH
\end{aligned}$$

$\overline{CreditCheckNotOK}_{c,i,o} + \overline{CreditCheckOK}_{c,i,o}$ represents the non-deterministic choice between the negative and positive outcomes of `Credit Check`. The negative outcome results in the following elaboration, which terminates the workflow:

$$\begin{aligned}
&\xrightarrow{\tau} WORKFLOW \mid \overline{Reject}_{c,i,o} \mid \\
&\quad SHIP \mid BILL \mid ARC \mid ARCH \\
&\xrightarrow{\overline{Reject}_{c,i,o}} WORKFLOW \mid 0 \mid SHIP \mid BILL \mid ARC \mid ARCH
\end{aligned}$$

The positive outcome of `Credit Check` results in the following elaboration:

$$\begin{aligned}
&\xrightarrow{\tau} WORKFLOW \mid \overline{Ship}_{c,i,o} \mid SHIP \mid BILL \mid ARC \mid ARCH \\
&\xrightarrow{\tau} WORKFLOW \mid \tau.\overline{Bill}_{c,i,o} \mid BILL \mid ARC \mid ARCH \\
&\xrightarrow{\tau} WORKFLOW \mid \overline{Bill}_{c,i,o} \mid BILL \mid ARC \mid ARCH \\
&\xrightarrow{\tau} WORKFLOW \mid \tau.\overline{Archive}_{c,i,o} \mid ARC \mid ARCH \\
&\xrightarrow{\tau} WORKFLOW \mid \overline{Archive}_{c,i,o} \mid ARC \mid ARCH
\end{aligned}$$

$\xrightarrow{\tau}$ $WORKFLOW \mid \tau.\overline{ArchiveOK}_{c,i,o} \mid ARCH$

$\xrightarrow{\tau}$ $WORKFLOW \mid \overline{ArchiveOK}_{c,i,o} \mid ARCH$

$\xrightarrow{\tau}$ $WORKFLOW \mid \overline{Confirm}_{c,i,o}$

$\xrightarrow{\overline{Confirm}_{c,i,o}}$ $WORKFLOW \mid 0$

Thus, Requirement B.4 on Configuration 1 is satisfied, and the workflow terminates.

Similarly, we can show the elaboration of *WORKFLOW'* always reaches 0, which implies a workflow with Configuration 2 terminates.

We represent the termination of a reconfiguration of a workflow as the disappearance of the fraction process instances used to represent the reconfiguration of the workflow. Therefore, the reconfiguration of a workflow with Configuration 1 terminates if the fraction process instances in *RM* used to reconfigure the *WORKFLOW* instance that represents the workflow disappear. We show the elaboration of *WORKFLOW* | *RM* can result in the disappearance of the fraction processes in *RM* used to reconfigure *WORKFLOW*:

WORKFLOW | *RM*

$\xrightarrow{Receipt_{c,i,o}}$ $WORKFLOW \mid \overline{InventoryCheck}_{c,i,o} \mid IC \mid ICH \mid CC \mid CCH \mid SHIP \mid BILL \mid ARC \mid ARCH \mid RM$

$\xrightarrow{trigger1}$ $WORKFLOW \mid \overline{InventoryCheck}_{c,i,o} \mid IC \mid ICH \mid CC \mid CCH \mid SHIP \mid BILL \mid ARC \mid ARCH \mid \frac{WORKFLOW'}{WORKFLOW} \mid RM$

$\xrightarrow{\tau}$ $\overline{InventoryCheck}_{c,i,o} \mid IC \mid ICH \mid CC \mid CCH \mid SHIP \mid BILL \mid ARC \mid ARCH \mid WORKFLOW' \mid RM$

Thus, the reconfiguration of an executing workflow with Configuration 1 is initiated, and proceeds by reconfiguring the non-executing workflow with Configuration 1 to a workflow with Configuration 2. Hence, Requirement D.3 on the reconfiguration is satisfied.

$\xrightarrow{\tau}$ $\tau.(\overline{InventoryCheckNotOK}_{c,i,o} + \overline{InventoryCheckOK}_{c,i,o}) \mid ICH \mid CC \mid CCH \mid SHIP \mid BILL \mid ARC \mid ARCH \mid WORKFLOW' \mid RM$

$\xrightarrow{\tau}$ $\overline{InventoryCheckNotOK}_{c,i,o} + \overline{InventoryCheckOK}_{c,i,o} \mid ICH \mid CC \mid CCH \mid SHIP \mid BILL \mid ARC \mid ARCH \mid WORKFLOW' \mid RM$

$\xrightarrow{trigger2}$ $\overline{InventoryCheckNotOK}_{c,i,o} + \overline{InventoryCheckOK}_{c,i,o} \mid ICH \mid CC \mid CCH \mid SHIP \mid BILL \mid ARC \mid ARCH \mid WORKFLOW' \mid \frac{ICH'}{ICH} \mid \frac{CCH'}{CCH} \mid \frac{SHIP'}{SHIP} \mid \frac{BILL'}{BILL} \mid \frac{ARC'}{ARC} \mid \frac{0}{ARCH} \mid RM$

$\xrightarrow{\tau}$ $\overline{InventoryCheckNotOK}_{c,i,o} + \overline{InventoryCheckOK}_{c,i,o} \mid CC \mid CCH \mid SHIP \mid BILL \mid ARC \mid ARCH \mid WORKFLOW' \mid ICH' \mid \frac{CCH'}{CCH} \mid \frac{SHIP'}{SHIP} \mid \frac{BILL'}{BILL} \mid \frac{ARC'}{ARC} \mid \frac{0}{ARCH} \mid RM$

Thus, the executing workflow with Configuration 1 is allowed to continue its execution until its next activity (the decision procedure after `Inventory Check`) is different from that of a workflow with Configuration 2, at which point the activity is replaced by corresponding activities (including `Supplier Check`) of a workflow with Configuration 2. Hence, Requirement D.1 on the reconfiguration is satisfied. The negative outcome of `Inventory Check` and the positive outcome of `Supplier Check` for order *o* results in the following elaboration:

$\xrightarrow{\tau}$ $CC \mid CCH \mid SHIP \mid BILL \mid ARC \mid ARCH \mid WORKFLOW' \mid \overline{SupplierCheck}_{c,i,o} \mid SC \mid \frac{CCH'}{CCH} \mid \frac{SHIP'}{SHIP} \mid \frac{BILL'}{BILL} \mid \frac{ARC'}{ARC} \mid \frac{0}{ARCH} \mid RM$

$\xrightarrow{\overline{SupplierCheck}_{c,i,o}}$ $CC \mid CCH \mid SHIP \mid BILL \mid ARC \mid ARCH \mid WORKFLOW' \mid SC \mid \frac{CCH'}{CCH} \mid \frac{SHIP'}{SHIP} \mid \frac{BILL'}{BILL} \mid \frac{ARC'}{ARC} \mid \frac{0}{ARCH} \mid RM$

$\xrightarrow{SupplierCheckOK_{c,i,o}}$ $CC \mid CCH \mid SHIP \mid BILL \mid ARC \mid ARCH \mid WORKFLOW' \mid \overline{CreditCheck}_{c,i,o} \mid \frac{CCH'}{CCH} \mid \frac{SHIP'}{SHIP} \mid \frac{BILL'}{BILL} \mid \frac{ARC'}{ARC} \mid \frac{0}{ARCH} \mid RM$

$\xrightarrow{\tau}$ $\tau.(\overline{CreditCheckNotOK}_{c,i,o} + \overline{CreditCheckOK}_{c,i,o}) \mid CCH \mid SHIP \mid BILL \mid ARC \mid ARCH \mid WORKFLOW' \mid \frac{CCH'}{CCH} \mid \frac{SHIP'}{SHIP} \mid \frac{BILL'}{BILL} \mid \frac{ARC'}{ARC} \mid \frac{0}{ARCH} \mid RM$

$\xrightarrow{\tau}$ $\overline{CreditCheckNotOK}_{c,i,o} + \overline{CreditCheckOK}_{c,i,o} \mid CCH \mid SHIP \mid BILL \mid ARC \mid ARCH \mid WORKFLOW' \mid \frac{CCH'}{CCH} \mid \frac{SHIP'}{SHIP} \mid \frac{BILL'}{BILL} \mid \frac{ARC'}{ARC} \mid \frac{0}{ARCH} \mid RM$

$\xrightarrow{\tau}$ $\overline{CreditCheckNotOK}_{c,i,o} + \overline{CreditCheckOK}_{c,i,o} \mid SHIP \mid BILL \mid ARC \mid ARCH \mid WORKFLOW' \mid CCH' \mid \frac{SHIP'}{SHIP} \mid \frac{BILL'}{BILL} \mid \frac{ARC'}{ARC} \mid \frac{0}{ARCH} \mid RM$

Thus, the partially reconfigured workflow is allowed to continue its execution until its next activity (the decision procedure after `Credit Check`) has to be reconfigured to enable the concurrent execution of `Billing` and `Shipping` in order to satisfy Requirement C.5.b on Configuration 2. The positive outcome of `Credit Check` for order *o* results in the following elaboration:

$\xrightarrow{\tau}$ $SHIP \mid BILL \mid ARC \mid ARCH \mid WORKFLOW' \mid \overline{Ship}_{c,i,o} \mid \overline{Bill}_{c,i,o} \mid \frac{SHIP'}{SHIP} \mid \frac{BILL'}{BILL} \mid \frac{ARC'}{ARC} \mid \frac{0}{ARCH} \mid RM$

$\xrightarrow{\tau}$ $BILL \mid ARC \mid ARCH \mid WORKFLOW' \mid \overline{Ship}_{c,i,o} \mid \overline{Bill}_{c,i,o} \mid SHIP' \mid \frac{BILL'}{BILL} \mid \frac{ARC'}{ARC} \mid \frac{0}{ARCH} \mid RM$

$\xrightarrow{\tau}$ $ARC \mid ARCH \mid WORKFLOW' \mid \overline{Ship}_{c,i,o} \mid \overline{Bill}_{c,i,o} \mid SHIP' \mid BILL' \mid \frac{ARC'}{ARC} \mid \frac{0}{ARCH} \mid RM$

$\xrightarrow{\tau}$ $ARC \mid ARCH \mid WORKFLOW' \mid \overline{Ship}_{c,i,o} \mid SHIP' \mid \tau.\overline{BillOK}_{c,i,o} \mid \frac{ARC'}{ARC} \mid \frac{0}{ARCH} \mid RM$

$$\xrightarrow{\tau} \quad ARC \mid ARCH \mid WORKFLOW' \mid \tau.\overline{ShipOK}_{c,i,o} \mid \tau.\overline{BillOK}_{c,i,o} \mid \frac{ARC'}{ARC} \mid \frac{0}{ARCH} \mid RM$$

$$\xrightarrow{\tau} \quad ARC \mid ARCH \mid WORKFLOW' \mid \tau.\overline{ShipOK}_{c,i,o} \mid \overline{BillOK}_{c,i,o} \mid \frac{ARC'}{ARC} \mid \frac{0}{ARCH} \mid RM$$

$$\xrightarrow{\tau} \quad ARC \mid ARCH \mid WORKFLOW' \mid \overline{ShipOK}_{c,i,o} \mid \overline{BillOK}_{c,i,o} \mid \frac{ARC'}{ARC} \mid \frac{0}{ARCH} \mid RM$$

$$\xrightarrow{\tau} \quad ARCH \mid WORKFLOW' \mid \overline{ShipOK}_{c,i,o} \mid \overline{BillOK}_{c,i,o} \mid ARC' \mid \frac{0}{ARCH} \mid RM$$

$$\xrightarrow{\tau} \quad ARCH \mid WORKFLOW' \mid \overline{ShipOK}_{c,i,o} \mid ShipOK_{c,i,o}.\tau \mid \frac{0}{ARCH} \mid RM$$

$$\xrightarrow{\tau} \quad ARCH \mid WORKFLOW' \mid \tau \mid \frac{0}{ARCH} \mid RM$$

$$\xrightarrow{\tau} \quad ARCH \mid WORKFLOW' \mid \frac{0}{ARCH} \mid RM$$

$$\xrightarrow{\tau} \quad WORKFLOW' \mid 0 \mid RM$$

Thus, the reconfigured workflow terminates, Requirement D.2 on the reconfiguration is satisfied, and the reconfiguration of the workflow terminates.

The above elaboration was produced manually, and reconfiguration transitions were performed when **not** to do so would break a requirement. However, this method of reconfiguration is not enforced by the semantics of basic CCS^{dp}. For example, the following elaboration of $WORKFLOW \mid RM$ begins as above and progresses to:

$$\overline{InventoryCheckNotOK}_{c,i,o} + \overline{InventoryCheckOK}_{c,i,o} \mid ICH \mid CC \mid CCH \mid SHIP \mid BILL \mid ARC \mid ARCH \mid WORKFLOW' \mid \frac{ICH'}{ICH} \mid \frac{CCH'}{CCH} \mid \frac{SHIP'}{SHIP} \mid \frac{BILL'}{BILL} \mid \frac{ARC'}{ARC} \mid \frac{0}{ARCH} \mid RM$$

but the workflow terminates prematurely in Configuration 1 and its reconfiguration fails to terminate (i.e. deadlocks) due to non-determinism of the transitions:

$$\xrightarrow{\tau} \quad \overline{Reject}_{c,i,o} \mid CC \mid CCH \mid SHIP \mid BILL \mid ARC \mid ARCH \mid WORKFLOW' \mid \frac{ICH'}{ICH} \mid \frac{CCH'}{CCH} \mid \frac{SHIP'}{SHIP} \mid \frac{BILL'}{BILL} \mid \frac{ARC'}{ARC} \mid \frac{0}{ARCH} \mid RM$$

$$\xrightarrow{\overline{Reject}_{c,i,o}} \quad 0 \mid CC \mid CCH \mid SHIP \mid BILL \mid ARC \mid ARCH \mid WORKFLOW' \mid \frac{ICH'}{ICH} \mid \frac{CCH'}{CCH} \mid \frac{SHIP'}{SHIP} \mid \frac{BILL'}{BILL} \mid \frac{ARC'}{ARC} \mid \frac{0}{ARCH} \mid RM$$

E. Extensions

Problematic non-deterministic transitions can be avoided by using a priority scheme for transitions that is designed to satisfy requirements on workflows and on their reconfiguration. Since the requirements can be application specific, different priority schemes may be necessary. Therefore, the semantics of basic CCS^{dp} should be extended with a generic notion of transition priority such that different system models can be produced using different priority schemes.

Section V-D describes the unplanned reconfiguration of a single executing workflow. In order to describe the unplanned reconfiguration of multiple executing workflows, a dynamic binding between *WORKFLOW* instances and *RM* instances is necessary that will support the selective reconfiguration of specific process instances. Such a binding can be achieved by extending the semantics of process matching to use process identifiers. If a process identifier is passed as a parameter to a fraction process, the fraction can reconfigure different process instances in a flexible and controlled manner. Furthermore, the identification of a specific process for reconfiguration precludes the matching of other processes, and thereby significantly reduces the computational complexity of matching. For example, in the expression $p_1 \mid p_2 \mid p_3 \mid x(i).\frac{p'}{p}(i)$ where p_1, p_2, and p_3 are different instances of the same process p, $\frac{p'}{p}(1)$ will be able to reconfigure only p_1, and the processes p_2, p_3, $p_1 \mid p_2$, $p_1 \mid p_3$, $p_2 \mid p_3$, and $p_1 \mid p_2 \mid p_3$ will not be tested for matching. It is important to notice that basic CCS^{dp} is a class-based process algebra. That is, like numbers in arithmetic, the processes in basic CCS^{dp} are classes, and different instances of a process can be used interchangeably in any context with identical results. However, the use of process identifiers in process matching makes the modification of basic CCS^{dp} an instance-based process algebra, so that different instances of a process with different identifiers in identical contexts can produce different results.

We briefly consider the reverse reconfiguration of a workflow (from Configuration 2 to Configuration 1) for the sake of completeness. The reconfiguration is performed by the reconfiguration manager denoted by the process *MR*:

$$MR \triangleq trigger4.\left(\frac{WORKFLOW}{WORKFLOW'} \mid MR\right) + trigger5.\left(\frac{ICH}{ICH'} \mid \frac{CCH}{CCH'} \mid \frac{SHIP}{SHIP'} \mid \frac{BILL}{BILL'} \mid \frac{ARC \mid ARCH}{ARC'} \mid MR\right)$$

Transposing Configuration 1 and Configuration 2 in the reconfiguration requirements defined in Section II-D, the reconfiguration from Configuration 2 to Configuration 1 is restricted by the existence of `Supplier Check` (see Figure 3 and Figure 2). If the outcome of `Inventory Check` is negative, `Supplier Check` is performed, which cannot be done in Configuration 1. Therefore, the reconfigured workflow cannot meet Requirement B.3 of Configuration 1. Hence, the reconfiguration cannot be performed. If the outcome

of `Inventory Check` is positive, the workflow can be reconfigured just after `Credit Check`. However, there is no mechanism in basic CCS^{dp} for testing the history of transitions of a workflow in order to determine a future transition. Such testing can be performed by extending the syntax of a process to include the history of transitions that produced the process, and extending the semantics of process matching to include matching of traces of transitions; and thereby increases the potential for reconfiguration of workflows.

F. Evaluation

We now identify the strengths and weaknesses of basic CCS^{dp} from the modelling of the case study workflow and its reconfiguration, and the verification of their requirements.

Strengths

Instances of software components and tasks can be modelled as process instances with different identifiers. Instances of communication links can be identified indirectly using process identifiers and port names that are unique to the linked processes.

The replacement, deletion, and creation of software components and tasks can be modelled using fraction processes. Both components and tasks are represented as processes, which can be identified for reconfiguration using the denominator of a fraction, and replaced by the numerator of the fraction, as demonstrated in Section V-D. Deletion of a process is expressed as replacement by 0 (the identity process). A process is created either by including it in the numerator of a fraction, or by using a guarded parallel composition of processes that spawns a new process after the guard is released (as in CCS).

Planned and unplanned reconfiguration can both be modelled using fraction processes. In modelling planned reconfiguration, the reconfiguring fractions are used within the system model, whereas for unplanned reconfiguration, the fractions are used in the context of the system model (see Section V-C).

The interleaving semantics of basic CCS^{dp} determines its concurrency properties. Thus, concurrent execution of application and reconfiguration tasks is modelled as the set of total orders of the transitions of processes, and functional interference between the tasks is modelled as interleaved computation and reconfiguration transitions of the processes (see Section V-D).

State transitions of software components and of tasks are modelled as process transitions.

Basic CCS^{dp} is a terse formalism for several reasons: CCS is terse, fraction processes do not contain implementation detail, overloading the parallel composition operator avoids the use of a new operator for performing reconfiguration (such as the interrupt operator in *CSP*), and process matching uses behaviour to match processes (although this is computationally complex).

Weaknesses

The termination of the reconfiguration process cannot be verified due to non-determinism of transitions (see Section V-D). However, this problem can be solved by using a priority scheme for transitions that is designed to satisfy the configuration and reconfiguration requirements of the workflow, as suggested in Section V-E. Basic CCS^{dp} can express the functional correctness requirements of application and reconfiguration tasks that can be represented as process congruences, and the requirements can be verified using equational reasoning [3]. However, such requirements are very limited in number due to the highly restrictive definition of process congruence in basic CCS^{dp} [3]. A more versatile approach to verification is model checking.

In order to reconfigure selectively specific process instances, it would be necessary to extend the semantics of process matching to use process identifiers, as suggested in Section V-E.

The creation and deletion of communication links between software components and between tasks can be expressed as process replacement using fraction processes, but the representation is clumsy. A simpler solution is to extend the syntax of basic CCS^{dp} processes to enable link passing, as in π-calculi [31].

Physical nodes are not modelled in basic CCS^{dp}. Hence, the relocation of software components or tasks on physical nodes cannot be modelled. However, if the process syntax is extended with a location attribute and the semantics of communication is extended with process passing, then relocation can be modelled simply as communication with process passing, in which the location of the process being passed changes from the location of the sending process to the location of the receiving process.

State transfer between software components and between tasks can be modelled by encoding the state in the names of the complementary communicating actions, or by replacing the receiving process with a process that has received the transferred state using a fraction process. However, a much simpler way

of modelling state transfer is to use value passing communication.

The preemption of actions is not modelled.

Currently, basic CCS^{dp} does not have tool support, due to the novel nature of process matching. However, we are modifying the formalism to reduce its limitations and there are plans to develop tools for modelling and verification.

In conclusion, basic CCS^{dp} tersely models planned and unplanned process reconfiguration and functional interference between application and reconfiguration tasks using the notion of the fraction process. The limitations of basic CCS^{dp} can be significantly reduced by allocating priorities to process transitions, extending the semantics of process matching to use process identifiers, expressing link and process passing communication, and providing tool support.

VI. Discussion and Conclusions

This paper has used the dynamic reconfiguration of a simple office workflow for order processing as a case study in order to compare the modelling and analysis capabilities of three different kinds of formalism, represented by VDM, CPOGs, and basic CCS^{dp}. The formalisms were used by different authors who worked independently of each other in order to follow the 'idioms' of their respective formalisms.

VDM was used to represent the workflow as a data type, specifically, as traces of actions, on which invariants and pre-conditions were defined that (if satisfied) would ensure the workflow requirements would be met. The execution of the workflow was performed by an interpreter. The reconfiguration of the workflow was expressed by passing Configuration 2 as a parameter to the interpreter during its execution of Configuration 1. The interpreter performed the reconfiguration if the trace executed so far in Configuration 1 was a prefix of a correct trace of Configuration 2, in order to meet Requirement D.2 of the reconfiguration. Concurrency was represented using interleaved actions within the interpreter and was not modelled explicitly.

CPOGs were used to represent each configuration of the workflow algebraically as a set of graphs of actions, where each vertex and arc of a graph could be annotated with a predicate. The graphs were transformed using the CPOG axioms, and the requirements on a configuration were verified by inspecting the result of the transformation. The reconfiguration of the workflow was determined using the notion of compatible history between the two configurations. Thus, Configuration 1 could be dynamically reconfigured to Configuration 2 if the graphs of actions of the two configurations were identical up to the point of reconfiguration, in order to meet Requirement D.2 of the reconfiguration. The use of graphs enables CPOGs to represent easily both workflow and reconfiguration actions that execute with true concurrency, although their interference was not modelled.

Basic CCS^{dp} was used to represent each configuration of the workflow as a process. The workflow was reconfigured by changing the process structure of its configuration, that is, by creating new processes and deleting or replacing existing processes of the configuration, through interactions with fraction processes located in the context of the process expression representing the workflow; thereby modelling abstractly the way in which a system is reconfigured through interactions with a patch located in its environment. Verification of the requirements of the workflow and its reconfiguration was done by inspecting the traces of the transitions of the processes representing the workflow and its reconfiguration respectively. Interference due to concurrency was modelled explicitly using interleaved workflow and reconfiguration transitions.

The evaluations of the three formalisms show that none of them is ideal, since none of them meets all the requirements on an ideal formalism for dynamic software reconfiguration defined in Section II-F. For example, key requirements include: the ability to express tersely change in the composition and structure of software components and tasks for both planned and unplanned dynamic reconfiguration; the ablility to express tersely the concurrent execution of tasks and their functional interference; and the ability to verify the functional correctness requirements of tasks, which includes verifying the functional correctness of refinements. Neither VDM, nor CPOGs, nor basic CCS^{dp} meets all three requirements. However, a combination of the formalisms does meet the requirements. Furthermore, all three formalisms can easily express traces of actions. Therefore, the formalisms are complementary, and it should be possible to combine them using basic CCS^{dp} for modelling, CPOGs for verification, and VDM for type checking and refinement.

The main strength of basic CCS^{dp} is its ability to model abstractly and tersely both planned and unplanned reconfiguration of concurrently executing tasks. Regarding its weaknesses: non-deterministic transitions can be controlled by using a priority scheme for transitions, specific process instances can be selectively reconfigured by using process identifiers in process matching, link reconfiguration and task

relocation can be modelled by using value passing communication in which links and processes can be passed (see Section V-F). The major weakness of basic CCS^{dp} is the highly restrictive definition of process congruence that severely limits its ability to verify requirements using equational reasoning [3]. In contrast, the main strength of CPOGs is their ability to verify requirements. It should be possible to convert a process in basic CCS^{dp} into a CPOG, since the elaboration of a process is an annotated graph. The CPOG can then be transformed into a canonical form in order to verify requirements. Furthermore, the CPOG will be amenable to model checking, since its annotations are predicates. However, neither basic CCS^{dp} nor CPOGs were designed to develop operational systems. In contrast, VDM was designed for formal development. As with CPOGs, it should be possible to convert a process in basic CCS^{dp} into a graph of actions in VDM for type checking and refinement into an executable form. Notice that VDM can pass functions as parameters to operations. Therefore, it should be possible to pass a predicate (expressing a pre-condition or an invariant) as well as a workflow as parameters to a reconfiguration operation (although this was not modelled). This is necessary because the invariants and pre-conditions that must be satisfied in order to ensure the correctness of a reconfiguration cannot always be pre-defined for a dynamically evolving system. Thus, it should be possible to construct an integrated approach to the formal modelling, verification, and development of dynamically reconfigurable dependable systems based on VDM, CPOGs, and basic CCS^{dp} or a combination of similar formalisms.

VII. Acknowledgements

This work was partly funded by the EPSRC under the terms of a graduate studentship, and by the TrAmS, Deploy, and UNCOVER projects. The authors acknowledge the help given by numerous colleagues, in particular: Jeremy Bryans, John Fitzgerald, Regina Frei, Kohei Honda, Alexei Iliasov, Cliff Jones, Maciej Koutny, Manuel Mazzara, Richard Payne, Traian Florin Serbanuta, Giovanna Di Marzo Serugendo, and Chris Woodford.

References

[1] J. P. A. Almeida, M. Wegdam, M. van Sinderen, and L. Nieuwenhuis, "Transparent dynamic reconfiguration for corba," in *Proceedings of the 3rd International Symposium on Distributed Objects and Applications*, 2001, pp. 197–207.

[2] D. Ardagna and B. Pernici, "Adaptive service composition in flexible processes," *IEEE Transactions on Software Engineering*, vol. 33, no. 6, pp. 369–384, 2007.

[3] A. Bhattacharyya, "Formal modelling and analysis of dynamic reconfiguration of dependable systems," Ph.D. dissertation, Newcastle University School of Computing Science, 2013.

[4] J. Bicarregui, J. Fitzgerald, P. Lindsay, R. Moore, and B. Ritchie, *Proof in VDM: A Practitioner's Guide*, ser. FACIT. Springer-Verlag, 1994, iSBN 3-540-19813-X.

[5] T. Bloom and M. Day, "Reconfiguration and module replacement in argus: theory and practice," *Software Engineering Journal (Special Issue)*, vol. 8, no. 2, pp. 102–108, 1993.

[6] J. S. Bradbury, J. R. Cordy, J. Dingel, and M. Wermelinger, "A survey of self-management in dynamic software architecture specifications," in *Proceedings of the 1st ACM SIGSOFT Workshop on Self-Managed Systems*, 2004, pp. 28–33.

[7] L. Coyle, M. Hinchey, B. Nuseibeh, and J. L. Fiadeiro, "Guest editors' introduction: Evolving critical systems," *IEEE Computer*, vol. 43, no. 5, pp. 28–33, 2010.

[8] N. Eén and N. Sörensson, "An Extensible SAT-solver," *Theory and Applications of Satisfiability Testing*, pp. 333–336, 2004.

[9] C. Ellis, K. Keddara, and G. Rozenberg, "Dynamic change within workflow systems," in *Proceedings of the Conference on Organizational Computing Systems*. ACM, 1995, pp. 10–21.

[10] S. Fischmeister and K. Winkler, "Non-blocking deterministic replacement of functionality, timing, and data-flow for hard real-time systems at runtime," in *Proceedings of the 17th Euromicro Conference on Real-Time Systems*. IEEE Computer Society, 2005, pp. 106–114.

[11] J. Fitzgerald, P. G. Larsen, P. Mukherjee, N. Plat, and M. Verhoef, *Validated Designs for Object–oriented Systems*. Springer, New York, 2005. [Online]. Available: http://www.vdmbook.com

[12] J. Fitzgerald, P. G. Larsen, and S. Sahara, "VDMTools: Advances in Support for Formal Modeling in VDM," *ACM Sigplan Notices*, vol. 43, no. 2, pp. 3–11, February 2008.

[13] E. Fărcaş, "Scheduling multi-mode real-time distributed components," Ph.D. dissertation, University of Salzburg Department of Computer Sciences, 2006.

[14] M. Hilario, P. Nguyen, H. Do, A. Woznica, and A. Kalousis, "Ontology-based meta-mining of knowledge discovery workflows," in *Meta-Learning in Computational Intelligence*, ser. Studies in Computational Intelligence, vol. 358. Springer-Verlag, 2011, pp. 273–315.

[15] P. Hudak, "Building domain-specific embedded languages," *ACM Computing Surveys*, vol. 28, no. 4, p. 196, 1996.

[16] "Information technology – Programming languages, their environments and system software interfaces – Vienna Development Method – Specification Language – Part 1: Base language," December 1996.

[17] C. B. Jones, *Systematic Software Development using VDM*, 2nd ed. Prentice Hall International, 1990.

[18] C. B. Jones, "The early search for tractable ways of reasonning about programs," *IEEE, Annals of the History of Computing*, vol. 25, no. 2, pp. 26–49, 2003.

[19] G. Karsai, F. Massacci, L. J. Osterweil, and I. Schieferdecker, "Evolving embedded systems," *IEEE Computer*, vol. 43, no. 5, pp. 34–40, 2010.

[20] J. Kramer and J. Magee, "The evolving philosophers problem: Dynamic change management," *IEEE Transactions on Software Engineering*, vol. 16, no. 11, pp. 1293–1306, 1990.

[21] P. G. Larsen, "Ten Years of Historical Development: "Bootstrapping" VDMTools," *Journal of Universal Computer Science*, vol. 7, no. 8, pp. 692–709, 2001.

[22] P. G. Larsen, N. Battle, M. Ferreira, J. Fitzgerald, K. Lausdahl, and M. Verhoef, "The Overture Initiative – Integrating Tools for VDM," *SIGSOFT Softw. Eng. Notes*, vol. 35,

no. 1, pp. 1–6, January 2010. [Online]. Available: http://doi.acm.org/10.1145/1668862.1668864

[23] P. G. Larsen, K. Lausdahl, and N. Battle, "Combinatorial Testing for VDM," in *Proceedings of the 2010 8th IEEE International Conference on Software Engineering and Formal Methods*, ser. SEFM '10. Washington, DC, USA: IEEE Computer Society, September 2010, pp. 278–285, ISBN 978-0-7695-4153-2. [Online]. Available: http://dx.doi.org/10.1109/SEFM.2010.32

[24] P. G. Larsen and W. Pawłowski, "The Formal Semantics of ISO VDM-SL," *Computer Standards and Interfaces*, vol. 17, no. 5–6, pp. 585–602, September 1995.

[25] C.-Y. Lee, "Representation of switching circuits by binary-decision programs," *Bell System Technical Journal*, vol. 38, no. 4, pp. 985–999, 1959.

[26] "The Maude project website," 2014, http://maude.cs.uiuc.edu/.

[27] M. Mazzara, F. Abouzaid, N. Dragoni, and A. Bhattacharyya, "Toward design, modelling and analysis of dynamic workflow reconfiguration - a process algebra perspective," in *Proceedings of the 8th International Workshop on Web Services and Formal Method (WSFM 2011)*, ser. Lecture Notes in Computer Science, vol. 7176. Springer-Verlag, 2012, pp. 64–78.

[28] N. Medvidovic and R. N. Taylor, "A classification and comparison framework for software architecture description languages," *IEEE Transactions on Software Engineering*, vol. 26, no. 1, pp. 70–93, 2000.

[29] T. Mens, J. Magee, and B. Rumpe, "Evolving software architecture descriptions of critical systems," *IEEE Computer*, vol. 43, no. 5, pp. 42–48, 2010.

[30] R. Milner, *Communication and Concurrency*. Prentice Hall International (U.K.) Limited, 1989.

[31] R. Milner, *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.

[32] A. Mokhov and V. Khomenko, "Algebra of Parameterised Graphs," *ACM Transactions on Embedded Computing*, vol. 13, no. 4s, 2014.

[33] A. Mokhov and A. Yakovlev, "Conditional Partial Order Graphs: Model, Synthesis and Application," *IEEE Transactions on Computers*, vol. 59, no. 11, pp. 1480–1493, 2010.

[34] A. Mokhov, "Conditional partial order graphs," Ph.D. dissertation, Newcastle University, 2009.

[35] A. Mokhov, M. Rykunov, D. Sokolov, and A. Yakovlev, "Design of processors with reconfigurable microarchitecture," *Journal of Low Power Electronics and Applications*, vol. 4, no. 1, pp. 26–43, 2014.

[36] J. Montgomery, "A model for updating real-time applications," *Real-Time Systems*, vol. 27, no. 2, pp. 169–189, 2004.

[37] M. Nielsen, G. D. Plotkin, and G. Winskel, "Petri nets, event structures and domains, part I," *Theoretical Computer Science*, vol. 13, pp. 85–108, 1981.

[38] P. S. M. Pedro, "Schedulability of mode changes in flexible real-time distributed systems," Ph.D. dissertation, University of York Department of Computer Science, 1999.

[39] D. Sangiorgi and D. Walker, *The π-calculus: A Theory of Mobile Processes*. Cambridge University Press, 2001.

[40] L. Sha, R. Rajkumar, J. Lehoczky, and K. Ramamritham, "Mode change protocols for priority-driven preemptive scheduling," *The Journal of Real-Time Systems*, vol. 1, no. 3, pp. 243–264, 1989.

[41] D. B. Stewart, R. A. Volpe, and P. K. Khosla, "Design of dynamically reconfigurable real-time software using port-based objects," *IEEE Transactions on Software Engineering*, vol. 23, no. 12, pp. 759–776, 1997.

[42] K. Tindell, A. Burns, and A. Wellings, "Mode changes in priority pre-emptively scheduled systems," in *Proceedings of the IEEE Real-Time Systems Symposium*, 1992, pp. 100–109.

[43] M. Verhoef, P. G. Larsen, and J. Hooman, "Modeling and Validating Distributed Embedded Real-Time Systems with VDM++," in *FM 2006: Formal Methods*, ser. Lecture Notes in Computer Science 4085, J. Misra, T. Nipkow, and E. Sekerinski, Eds. Springer-Verlag, 2006, pp. 147–162.

[44] M. A. Wermelinger, "Specification of software architecture reconfiguration," Ph.D. dissertation, University of Lisbon Department of Informatics, 1999.

[45] T. Yu and K. J. Lin, "Adaptive algorithms for finding replacement services in autonomic distributed business processes," in *Proceedings of the 7th International Symposium on Autonomous Decentralized Systems*. IEEE, 2005, pp. 427–434.